9 Burrows–Wheeler indexes

Consider the largest sequenced genome to date, the approximately $25.5 \cdot 10^9$ base-pair-long genome of the white spruce *Picea glauca*. The suffix array of this genome would take approximately $1630 \cdot 10^9$ bits of space, or approximately 204 gigabytes, if we represent it with 64-bit integers, and it would take approximately 110 gigabytes if we represent it with fixed-length bit-field arrays. The suffix tree of the same genome would occupy from three to five times more space. However, the genome itself can be represented in just $51 \cdot 10^9$ bits, or approximately 6.4 gigabytes, assuming an alphabet of size $\sigma = 4$: this is equivalent to $n \log \sigma$ bits, where n is the length of the string. The gaps between the sizes of the suffix tree, of the suffix array, and of the original string are likely to grow as more species are sequenced: for example, the unsequenced genome of the flowering plant *Paris japonica* is estimated to contain $150 \cdot 10^9$ base pairs. The size of metagenomic samples is increasing at an even faster pace, with files containing approximately $160 \cdot 10^9$ base pairs already in public datasets.

Burrows–Wheeler indexes are a space-efficient variant of suffix arrays and suffix trees that take $n \log \sigma (1 + o(1))$ bits for a genome of length n on an alphabet of size σ (or just about 10 gigabytes for *Picea glauca*), and that support a number of high-throughput sequencing analyses approximately as fast as suffix arrays and suffix trees. Recall from Section 2.2 that we use the term *succinct* for data structures that, like Burrows–Wheeler indexes, occupy $n \log \sigma (1 + o(1))$ bits. This chapter walks the reader through the key algorithmic concepts behind Burrows–Wheeler indexes, leaving applications to Part IV and Part V.

After describing the Burrows–Wheeler transform of a string and related data structures for counting and locating all the occurrences of a pattern, the focus shifts to the *bidirectional* Burrows–Wheeler index, a powerful data structure that allows one to enumerate all internal nodes of the suffix tree of a string in a small amount of space. The bidirectional index will be used heavily in the following chapters, but for concreteness a sampler of its flexibility will be provided in Section 9.7.3. The chapter covers also advanced topics, like the space-efficient construction of the Burrows–Wheeler transform of a string, and concludes with a unified description of Burrows–Wheeler indexes for labeled trees, directed acyclic graphs, and de Bruijn graphs, which will be the substrate of a number of applications in Part IV and V.

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

9.1 Burrows–Wheeler transform (BWT)

Given a string T, its Burrows–Wheeler transform can be seen as the permutation of T induced by scanning its suffix array sequentially, and by printing the character that precedes each position in the suffix array. More formally, we have the following definition.

DEFINITION 9.1 *Let $T = t_1 t_2 \cdots t_n$ be a string such that $t_i \in \Sigma = [1..\sigma]$ for $i \in [1..n-1]$ and $t_n = \#$, where $\# = 0$ is a character that does not belong to alphabet Σ. The* Burrows–Wheeler transform *of T, denoted by BWT_T, is the permutation $L[1..n]$ of T defined as follows:*

$$L[i] = \begin{cases} T[\mathsf{SA}_T[i] - 1] & \text{if } \mathsf{SA}[i] > 1, \\ t_n = \# & \text{if } \mathsf{SA}[i] = 1. \end{cases}$$

See Figure 9.1 for an illustration of the transform.

Alternatively, BWT_T can be defined in terms of cyclic shifts of T: see Insight 9.1. In what follows, we repeatedly use the following function on two integers i and n:

$$i \ (\mathrm{mod}_1 \ n) = \begin{cases} n & \text{if } i = 0, \\ i & \text{if } i \in [1..n], \\ 1 & \text{if } i = n + 1. \end{cases}$$

	SA_T		BWT_T	
1	16	1	C	#
2	1	2	#	AGAGCGAGAGCGCGC# = T
3	7	3	G	AGAGCGCGC#
4	3	4	G	AGCGAGAGCGCGC#
5	9	5	G	AGCGCGC#
6	15	6	G	C#
7	5	7	G	CGAGAGCGCGC#
8	13	8	G	CGC#
9	11	9	G	CGCGC#
10	6	10	C	GAGAGCGCGC#
11	2	11	A	GAGCGAGAGCGCGC#
12	8	12	A	GAGCGCGC#
13	14	13	C	GC#
14	4	14	A	GCGAGAGCGCGC#
15	12	15	C	GCGC#
16	10	16	A	GCGCGC#

Figure 9.1 The Burrows–Wheeler transform of string $T =$ AGAGCGAGAGCGCGC#, shown for clarity with the suffix array of T and with the corresponding set of lexicographically sorted suffixes of T

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

Insight 9.1 Burrows–Wheeler transform and cyclic shifts

The Burrows–Wheeler transform was originally defined in terms of the *cyclic shifts* of a string $T = t_1 \cdots t_n \in [1..\sigma]^+$. Indeed, assume that we build the n cyclic shifts of T, or equivalently strings $t_1 t_2 \cdots t_n$, $t_2 t_3 \cdots t_n t_1$, ..., $t_n t_1 \cdots t_{n-1}$, and assume that we sort them into lexicographic order. For concreteness, let $T =$ `mississippi`. After sorting the cyclic shifts of T, we get the following matrix of characters M:

i	M
1	`imississipp`
2	`ippimissis`
3	`issippimis`
4	`ississippim`
5	`mississippi`
6	`pimississip`
7	`ppimississi`
8	`sippimissis`
9	`sissippimis`
10	`ssippimissi`
11	`ssissippimi`

The *first column* F of M contains the characters of T in sorted order, and the *last column* L of M (highlighted with bold letters) contains the BWT of T, that is, the string `pssmipissii`. The suffix of T in each cyclic permutation is underlined. The original string `mississippi` is row $x_1 = 5$ of matrix M, and its right-shift `imississipp` is row $x_2 = 1$. Note that shifting `mississippi` to the right moves its last character $L[x_1] =$ `i` to the beginning of the next cyclic permutation `imississipp`, that is, $L[x_1] = F[x_2]$. It follows that row x_2 must belong to the interval of all sorted cyclic shifts that satisfy $F[x_2] =$ `i`: see Section 9.1 for an algorithm that computes x_2 using just x_1 and L. Once we know how to move from the row x_i of a cyclic permutation to the row x_{i+1} of its right shift, we can reconstruct T *from right to left*, by printing the sequence $L[x_n] \cdots L[x_2]L[x_1]$.

Note that, since the string of this example is not periodic, no two cyclic shifts are identical, therefore the lexicographic order between every pair of cyclic shifts is well defined. In general, appending an artificial character $\# = 0 \notin [1..\sigma]$ at the end of T ensures that all cyclic shifts of $T\#$ are distinct for any T, and moreover that sorting the cyclic shifts of $T\#$ coincides with sorting the suffixes of $T\#$.

A key feature of BWT_T is that it is *reversible*: given $\mathsf{BWT}_T = L$, one can reconstruct the unique T of which L is the Burrows–Wheeler transform. Indeed, let V and W be two suffixes of T such that V is lexicographically smaller than W, and assume that both V and W are preceded by character a in T. It follows that suffix aV is lexicographically smaller than suffix aW, thus there is a bijection between suffixes *preceded by* a and suffixes that *start with* a that preserves the relative order of such suffixes. Consider thus suffix $T[i..n]$, and assume that it corresponds to position p_i in SA_T. If $L[p_i] = a$ is the

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

kth occurrence of a in $L[1..p_i]$, then the suffix $T[i-1 \pmod{1}\ n)..n]$ that starts at the *previous position* in T must be the kth suffix that starts with a in SA_T, and its position p_{i-1} in SA_T must belong to the compact interval $\overleftarrow{a}$ that contains all suffixes that start with a. For historical reasons, the function that projects the position p_i in SA_T of a suffix $T[i..n]$ to the position p_{i-1} in SA_T of the *previous suffix in circular string order* $T[i-1 \pmod{1}\ n)..n]$ is called LF-*mapping* (see Insight 9.1) and it is defined as

$$\mathsf{LF}(i) = j, \text{ where } \mathsf{SA}[j] = \mathsf{SA}[i] - 1 \pmod{1}\ n). \tag{9.1}$$

In what follows we will use $\mathsf{LF}^{-1}(j) = i$ to denote the *inverse* of function LF, or equivalently the function that projects position j in SA onto the position i in SA such that $\mathsf{SA}[i] = \mathsf{SA}[j] + 1 \pmod{1}\ n)$. Once we know how a right-to-left movement on string T projects into a movement on SA_T, we can reconstruct T from right to left, by starting from the position $x_1 = 1$ of suffix # in SA_T and by printing the sequence $\#L[x_1]L[x_2]\ldots L[x_{n-1}] = \underline{T}$, where $x_{i+1} = \mathsf{LF}(x_i)$ for $i \in [1..n-2]$. Similarly, we can reconstruct T from left to right by starting from the position $x_1 = \mathsf{LF}^{-1}(1)$ of T in SA_T, that is, from the position of # in BWT_T, and by printing the sequence $L[x_2]\ldots L[x_n]\# = T$, where $x_{i+1} = \mathsf{LF}^{-1}(x_i)$ for $i \in [1..n-1]$. More generally, we have the following lemma.

LEMMA 9.2 *Given a length m and a position i in SA_T, we can reconstruct the substring $T[\mathsf{SA}_T[i]-m \pmod{1}\ n)..\mathsf{SA}_T[i]-1 \pmod{1}\ n)]$ by iterating $m-1$ times the function LF, and we can reconstruct the substring $T[\mathsf{SA}_T[i]..\mathsf{SA}_T[i]+m-1 \pmod{1}\ n)]$ by iterating m times the function LF^{-1}.*

Note that we can compute all possible values of the function LF *without using* SA_T: indeed, we can apply Lemma 8.5 on page 134 to stably sort the list of tuples $(L[1], 1, 1)$, $(L[2], 2, 2)$, …, $(L[n], n, n)$ in $O(\sigma + n)$ time. Let L' be the resulting sorted list: it is easy to see that $\mathsf{LF}(L'[i].v) = i$ for all $i \in [1..n]$. In the next section we describe how to compute LF without even building and storing L'.

9.2 BWT index

9.2.1 Succinct LF-mapping

Let again L be the Burrows–Wheeler transform of a string $T = t_1 \cdots t_n$ such that $t_i \in [1..\sigma]$ for all $i \in [1..n-1]$, and $t_n = \#$, and assume that we have an array $C[0..\sigma]$ that stores in $C[c]$ the number of occurrences in T of all characters strictly smaller than c, that is, the sum of the frequency of all characters in the set $\{\#, 1, \ldots, c-1\}$. Note that $C[0] = 0$, and that $C[c] + 1$ is the position in SA_T of the first suffix that starts with character c. Recall that in Section 9.1 we defined $\mathsf{LF}(i) = j$ as the mapping from the position i of the kth occurrence of character c in L to the kth position inside the interval of character $c = L[i]$ in SA_T. It follows that $\mathsf{LF}(i) = C[L[i]] + k$, where the value $k = \mathtt{rank}_{L[i]}(L, i)$ can be computed in $O(\log \sigma)$ time if we represent L using a wavelet tree (see Section 3.3). Computing $\mathsf{LF}^{-1}(i)$ amounts to performing $\mathtt{select}_c(i - C[c])$,

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

where $c \in [0..\sigma]$ is the character that satisfies $C[c] < i \leq C[c+1]$ and can be found in $O(\log \sigma)$ time by binary-searching array C, and `select` can be computed in $O(\log \sigma)$ time if we represent L using a wavelet tree (see Section 3.3). This setup of data structures is called a *BWT index*.

DEFINITION 9.3 *Given a string $T \in [1..\sigma]^n$, a* BWT index *on T is a data structure that consists of*

- $\mathsf{BWT}_{T\#}$*, encoded using a wavelet tree; and*
- *the integer array $C[0..\sigma]$, which stores in $C[c]$ the number of occurrences in T# of all characters strictly smaller than c.*

Combining Lemma 9.2 with the implementation of LF and of LF^{-1} provided by the BWT index, we get the following results, which will be useful in the following chapters.

LEMMA 9.4 *There is an algorithm that, given the BWT index of a string $T \in [1..\sigma]^n$, outputs the sequence $\mathsf{SA}^{-1}_{T\#}[1], \ldots, \mathsf{SA}^{-1}_{T\#}[n]$ or its reverse in $O(\log \sigma)$ time per value, in $O(n \log \sigma)$ total time, and in $O(\log n)$ bits of working space.*

Figure 9.2 illustrates Lemma 9.4.

LEMMA 9.5 *Suppose that the BWT index of a string $T \in [1..\sigma]^n$ has already been built. Given a position i in $\mathsf{SA}_{T\#}$ and a length m, we can extract in $O(m \log \sigma)$ total time and in $O(\log n)$ bits of working space the following strings:*

- $W_R = t_j \cdot t_{j+1 \ (\mathrm{mod}_1 \ n+1)} \cdot \cdots \cdot t_{j+m-1 \ (\mathrm{mod}_1 \ n+1)}$*; and*
- $W_L = t_{j-1 \ (\mathrm{mod}_1 \ n+1)} \cdot t_{j-2 \ (\mathrm{mod}_1 \ n+1)} \cdot \cdots \cdot t_{j-m \ (\mathrm{mod}_1 \ n+1)}$,

Algorithm 9.1: Counting the number of occurrences of a pattern in a string using backward search

Input: Pattern $P = p_1 p_2 \cdots p_m$, count array $C[0..\sigma]$, function $\mathtt{rank}_c(L, i)$ on the $\mathsf{BWT}_T = L$ of string $T = t_1 \cdots t_n$, where $t_i \in [1..\sigma]$ for all $i \in [1..n-1]$ and $t_n = \#$.

Output: Number of occurrences of P in T.

```
1  i ← m;
2  (sp, ep) ← (1, n);
3  while sp ≤ ep and i ≥ 1 do
4      c ← p_i;
5      sp ← C[c] + rank_c(L, sp − 1) + 1;
6      ep ← C[c] + rank_c(L, ep);
7      i ← i − 1;
8  if ep < sp then
9      return 0;
10 else
11     return ep − sp + 1;
```

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

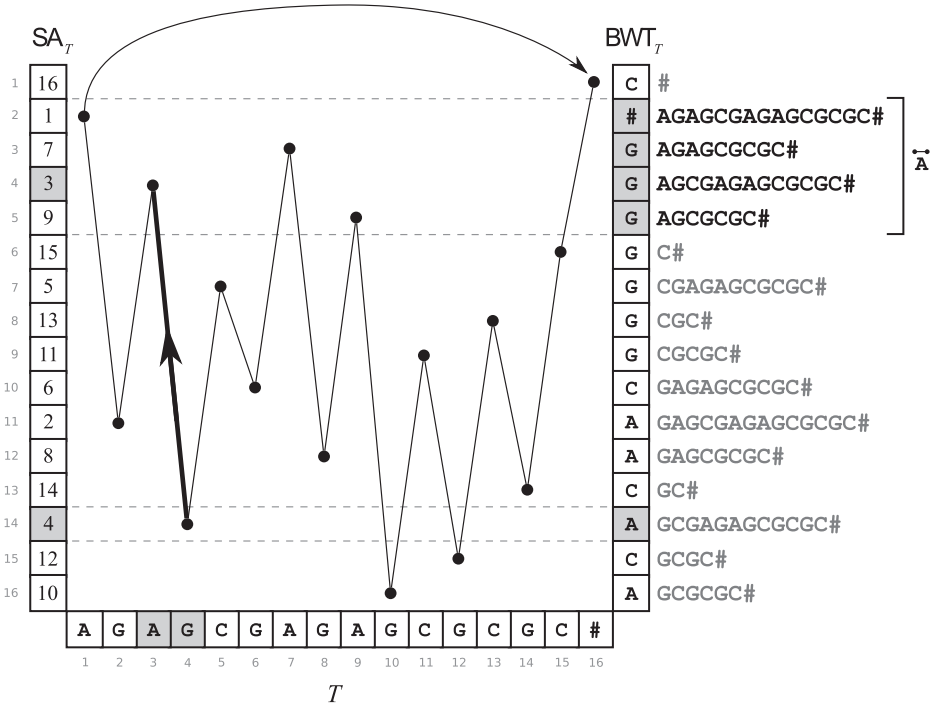

Figure 9.2 Function $\mathsf{LF}(i)$ for all $i \in [1..|T|]$, where $T =$ AGAGCGAGAGCGCGC#. Dots represent suffixes of T, displayed at the same time in string order (horizontal axis) and in suffix array order (vertical axis). Lines represent the function LF, which connects suffixes that are consecutive in string order. Consider for example suffix GCGAGAGCGCGC#, which starts at position 4 in T, and corresponds to position 14 in SA_T: since $\mathsf{BWT}_T[14] =$ A, and since there are three occurrences of A in $\mathsf{BWT}_T[1..14]$, it follows that the previous suffix AGCGAGAGCGCGC# that starts at position 3 in T is the third inside the interval of A in SA_T, therefore it corresponds to position 4 in SA_T. String T can be reconstructed from right to left by printing the characters that are met in BWT_T while following all arcs in the diagram, starting from position one in BWT_T.

where $j = \mathsf{SA}_{T\#}[i]$. *The characters of* W_R *and of* W_L *are extracted one by one in left-to-right order, in* $O(\log \sigma)$ *time per character.*

9.2.2 Backward search

Using the function LF, one can immediately implement Algorithm 9.1, which counts the number of occurrences in $T \in [1..\sigma]^{n-1}\#$ of a given pattern P, in $O(|P|)$ steps. Its correctness can be proved by induction, by showing that, at each step $i \in [1..|P|]$, the interval $[sp..ep]$ in the suffix array SA_T corresponds to all suffixes of T that are prefixed by $P[i..|P|]$ (see Figure 9.3). Algorithm 9.1 takes $O(|P|)$ time if the function `rank` can be computed in constant time on the Burrows–Wheeler transform of T: recall from Section 3.3 on page 24 that we can build a representation of BWT_T that supports `rank` queries in constant time using $\sigma|T|(1 + o(1))$ bits of space, or alternatively we can build a representation of BWT_T that supports `rank` queries in $O(\log \sigma)$ time using $|T|\log\sigma(1 + o(1))$ bits of space.

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

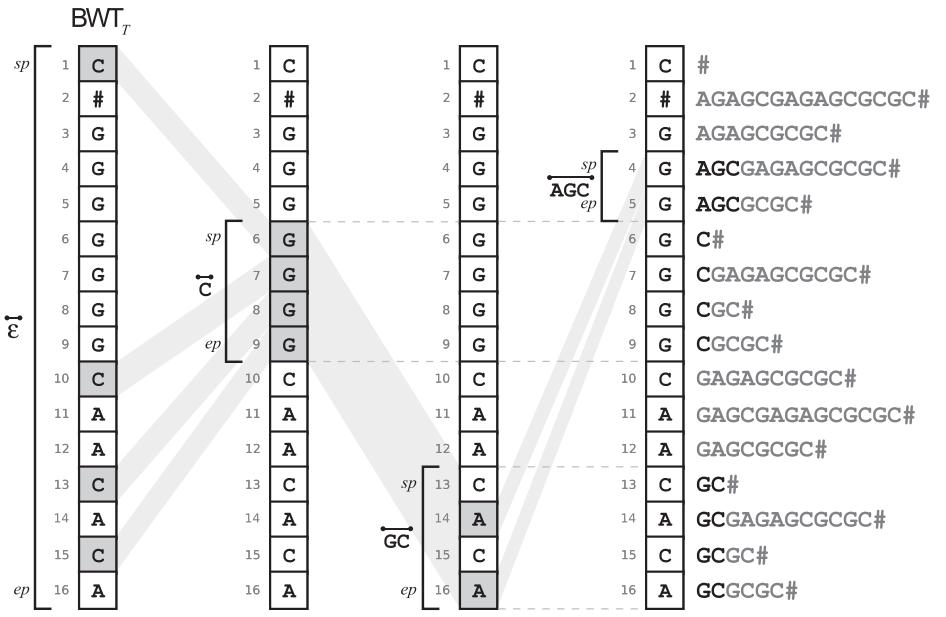

Figure 9.3 The backward search algorithm run on pattern $P = \texttt{AGC}$ in the Burrows–Wheeler transform of string $T = \texttt{AGAGCGAGAGCGCGC\#}$ (see Algorithm 9.1). Assume that we are in the interval [13..16] of string GC: the first occurrence of A inside this interval is at position 14, and it is preceded by two other occurrences of A in BWT_T. It follows that $\mathsf{LF}(14) = 4$, that is, the third position in the interval of character A in BWT_T. The last occurrence of A in the interval of GC is at position 16, and it is preceded by three other occurrences of A in BWT_T. It follows that $\mathsf{LF}(16) = 5$, the fourth position in the interval of character A. See Figure 8.1 for a similar search performed on the suffix array of T.

9.2.3 Succinct suffix array

So far we have described only how to support *counting* queries, and we are still not able to *locate* the starting positions of a pattern P in string $T \in [1..\sigma]^{n-1}\#$. One way to do this is to *sample* suffix array values, and to extract the missing values using the LF-mapping. Adjusting the sampling rate r gives different space/time tradeoffs.

Specifically, we sample all the values of $\mathsf{SA}[i]$ that satisfy $\mathsf{SA}[i] = rk$ for $0 \leq k \leq n/r$, and we store such samples consecutively, in the same order as in SA_T, in the array $\texttt{samples}[1..\lceil n/r \rceil]$. Note that this is equivalent to sampling every r positions *in string order*. We also mark in a bitvector $B[1..n]$ the positions of the suffix array that have been sampled, that is, we set $B[i] = 1$ if $\mathsf{SA}[i] = rk$, and we set $B[i] = 0$ otherwise. Then, if we are at a position i in SA_T such that $B[i] = 1$, we can extract the value of $\mathsf{SA}_T[i]$ by accessing $\texttt{samples}[\texttt{rank}_1(B, i)]$. Otherwise, if $B[i] = 0$, we can set an auxiliary variable j to i, and then iteratively reset j to $\mathsf{LF}(j) = C[L[j]] + \texttt{rank}_{L[j]}(L, j)$ until $B[j] = 1$, where L is again the Burrows–Wheeler transform of T: this corresponds to moving from right to left along string T until we find a sampled position. If we performed d such iterations of LF-mapping, we know that $\mathsf{SA}_T[i] = \mathsf{SA}_T[j] + d$.

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

Computing $\mathsf{SA}[i]$ with this strategy clearly takes $O(r \log \sigma)$ time, since each of the $d \leq r$ steps requires one `rank` computation on L, which can be answered in $O(\log \sigma)$ time if L is represented as a wavelet tree. One can set $r = \log^{1+\epsilon} n / \log \sigma$ for any given $\epsilon > 0$ to have the samples fit in $(n/r)\log n = n \log \sigma / \log^{\epsilon} n = o(n \log \sigma)$ bits, which is asymptotically the same as the space required for supporting counting queries. This setting implies that the extraction of $\mathsf{SA}[i]$ takes $O(\log^{1+\epsilon} n)$ time. Bitvector B and its supporting data structure for `rank` require $n + o(n)$ bits if implemented as described in Section 3.2. The resulting collection of data structures is called a *succinct suffix array*.

We leave the details of building the succinct suffix array of a string T from the BWT index of T to Exercise 9.2, and we just summarize the time and space complexities of the construction in the following theorem.

THEOREM 9.6 *The succinct suffix array of a string $T \in [1..\sigma]^n$ can be built from the BWT index of T in $O(n \log \sigma)$ time and $O(n \log \sigma + \mathtt{out})$ bits of space, where* `out` *is the size of the succinct suffix array.*

Succinct suffix arrays can be further extended into *self-indexes*. A self-index is a succinct representation of a string T that, in addition to supporting count and locate queries on arbitrary strings provided in input, allows one to access any substring of T by specifying its starting and ending position. In other words, a self-index for T completely replaces the original string T, which can be discarded.

Recall from Section 9.1 that we can reconstruct the whole string $T \in [1..\sigma]^{n-1}\#$ from BWT_T by applying the function LF iteratively. To reconstruct arbitrary substrings efficiently, it suffices to store, for every sampled position ri in string T, the position of suffix $T[ri..n]$ in SA_T: specifically, we use an additional array $\mathtt{pos2rank}[1..\lceil n/r \rceil]$ such that $\mathtt{pos2rank}[i] = j$ if $\mathsf{SA}_T[j] = ri$. Note that `pos2rank` can itself be seen as a sampling of the *inverse suffix array* at positions ri. Clearly, `pos2rank` takes the same amount of space as the array `samples`. Given an interval $[e..f]$ in string T, we can use $\mathtt{pos2rank}[k+1]$ to go to the position i of suffix $T[rk..n]$ in SA_T, where $k = \lceil f/r \rceil$ and rk is the smallest sampling position bigger than f in T. We can then apply LF-mapping $rk - e - 1$ times starting from i: the result is the whole substring $T[e..rk]$ printed from right to left, thus we can return its proper prefix $T[e..f]$. The running time of this procedure is $O((f - e + r)\log\sigma)$.

Making a succinct suffix array a self-index does not increase its space complexity asymptotically. We can thus define the succinct suffix array as follows.

DEFINITION 9.7 *Given a string $T \in [1..\sigma]^n$, we define the* succinct suffix array *of T as a data structure that takes $n \log \sigma (1 + o(1)) + O(n/r \log n)$ bits of space, where r is the sampling rate, and that supports the following queries.*

- `count`(P)*: return the number of occurrences of string $P \in [1..\sigma]^m$ in T in $O(m \log \sigma)$ time.*
- `locate`(i)*: return $\mathsf{SA}_{T\#}[i]$ in $O(r \log \sigma)$ time.*
- `substring`(e,f)*: return $T[e..f]$ in $O\Big(\big((f-e)+r\big)\log\sigma\Big)$ time.*

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

9.2.4 Batched locate queries

Recall from Section 9.2.3 that the succinct suffix array of a string $T \in [1..\sigma]^{n-1}$ allow one to translate a given position i in $\mathsf{BWT}_{T\#}$ into the corresponding position $\mathsf{SA}_{T\#}[i]$ in $O(r \log \sigma)$ time, where r is the sampling rate of the succinct suffix array. Performing $\mathtt{occ}$ such conversions would take $O(\mathtt{occ} \cdot r \log \sigma)$ time, which might be too much for some applications. In this section we describe an algorithm that takes $O(n \log \sigma + \mathtt{occ})$ time and $O(\mathtt{occ} \cdot \log n)$ bits of space, and that answers a set of $\mathtt{occ}$ locate queries *in batch* using the BWT index of T, rather than the succinct suffix array of T.

Specifically, assume that an algorithm $\mathcal{A}$ selects some (possibly repeated) positions $i_1, i_2, \ldots, i_{\mathtt{occ}}$ in $\mathsf{SA}_{T\#}$, converts them into positions of T, and prints the set of pairs $\{(\mathsf{SA}_{T\#}[i_k], \mathtt{data}_k) : k \in [1..\mathtt{occ}]\}$, where $\mathtt{data}_k$ is an encoding of application-specific data that $\mathcal{A}$ associates with position i_k. We can rewrite algorithm $\mathcal{A}$ as follows. We use a bitvector $\mathtt{marked}[1..n+1]$ to mark the positions in $\mathsf{SA}_{T\#}$ which must be converted to positions in T, and we use a buffer $\mathtt{pairs}[1..N]$ with $N \geq \mathtt{occ}$ to store a representation of the output of $\mathcal{A}$. Whenever $\mathcal{A}$ computes $\mathsf{SA}_{T\#}[i_k]$, we modify the algorithm so that it sets instead $\mathtt{marked}[i_k] = 1$. Whenever $\mathcal{A}$ prints the pair $(\mathsf{SA}_{T\#}[i_k], \mathtt{data}_k)$ to the output, we modify the algorithm so that it appends instead pair (i_k, p_k) to $\mathtt{pairs}$, where p_k is an integer that $\mathcal{A}$ uses to identify $\mathtt{data}_k$ uniquely.

Then, we invert $\mathsf{BWT}_{T\#}$ in $O(n \log \sigma)$ time, as described in Sections 9.1 and 9.2.1. During this process, whenever we are at a position i in $\mathsf{BWT}_{T\#}$, we also know the corresponding position $\mathsf{SA}_{T\#}[i]$ in $T\#$: if $\mathtt{marked}[i] = 1$, we append pair $(i, \mathsf{SA}_{T\#}[i])$ to an additional buffer $\mathtt{translate}[1..N]$ such that $N \geq \mathtt{occ}$. At the end of this process, the pairs in $\mathtt{translate}$ are in reverse string order, and the pairs in $\mathtt{pairs}$ are in arbitrary order: we thus sort both $\mathtt{translate}$ and $\mathtt{pairs}$ by suffix array position. Finally, we perform a linear, simultaneous scan of the two sorted arrays, matching a pair (i_k, p_k) in $\mathtt{pairs}$ to a corresponding pair $(i_k, \mathsf{SA}_{T\#}[i_k])$ in $\mathtt{translate}$, and printing $(\mathsf{SA}_{T\#}[i_k], \mathtt{data}_k)$ to the output by using p_k. See Algorithm 9.2 for the pseudocode of these steps.

Recall from Lemma 8.5 that we can sort a list of m triplets (p, s, v) by their primary key, breaking ties according to their secondary key, in $O(u + m)$ time and space, where $p \in [1..u]$ is a primary key, $s \in [1..u]$ is a secondary key, and v is an application-specific value. We can thus apply the radix sort algorithm of Lemma 8.5 to the array $\mathtt{pairs}[1..\mathtt{occ}]$, by interpreting each pair (i_k, p_k) as a triple $(\mathtt{msb}(i_k), \mathtt{lsb}(i_k), p_k)$, where $\mathtt{msb}(x)$ is a function that returns the most significant $\lceil (\log n)/2 \rceil$ bits of x and $\mathtt{lsb}(x)$ is a function that returns the least significant $\lfloor (\log n)/2 \rfloor$ bits of x. Since the resulting primary and secondary keys belong to the range $[1..2\sqrt{n}]$, sorting $\mathtt{pairs}$ takes $O(2\sqrt{n} + \mathtt{occ})$ time and space, and sorting $\mathtt{translate}$ takes $O(2\sqrt{n} + n) = O(n)$ time and space. We have thus proved the following lemma.

LEMMA 9.8 *Given the* **BWT** *index of a string $T = t_1 \cdots t_n$, where $t_i \in [1..\sigma]$ for all $i \in [1..n]$, a list $\mathtt{pairs}[1..\mathtt{occ}]$ of pairs (i_k, p_k), where $i_k \in [1..n+1]$ is a position in $\mathsf{SA}_{T\#}$ and p_k is an integer for all $k \in [1..\mathtt{occ}]$, and a bitvector $\mathtt{marked}[1..n+1]$, where $\mathtt{marked}[i_k] = 1$ for all i_k that appear in $\mathtt{pairs}$, we can transform every pair*

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

$(i_k, p_k) \in$ `pairs` *into the corresponding pair* $(\mathsf{SA}_{T\#}[i_k], p_k)$*, possibly altering the order of pairs in list* `pairs`*, in* $O(n \log \sigma + \mathtt{occ})$ *time and* $O(\mathtt{occ} \cdot \log n)$ *space.*

Exercise 9.3 explores an alternative batch strategy that does away with radix sort.

Algorithm 9.2: Answering multiple locate queries in batch. We assume that `radixSort`(X, y) is a function that sorts a list of pairs X by component y. Symbols p and s in the code refer to the primary and secondary key of a pair, respectively.

Input: List `pairs` containing `occ` pairs (i_k, p_k), where $i_k \in [1..n]$ is a position in SA_T for all $k \in [1..\mathtt{occ}]$. Bitvector `marked`$[1..n]$, where `marked`$[i_k] = 1$ for all i_k that appear at least once in `pairs`. Burrows–Wheeler transform of a string $T = t_1 \cdots t_n$ such that $t_i \in [1..\sigma]$ for all $i \in [1..n-1]$ and $t_n = \#$, with count array $C[0..\sigma]$ and support for $\mathtt{rank}_c$ operations, $c \in [0..\sigma]$.

Output: List `pairs`, possibly permuted, in which i_k is replaced by $\mathsf{SA}_T[i_k]$ for all $k \in [1..\mathtt{occ}]$.

```
1  i ← 1;
2  translate ← ∅;
3  for j ← n to 1 do
4      if marked[i] = 1 then
5          translate.append((i, j));
6      i ← LF(i);
7  radixSort(translate, 1);
8  radixSort(pairs, 1);
9  i ← 1;
10 j ← 1;
11 while i ≤ occ do
12     if pairs[i].p = translate[j].p then
13         pairs[i].p ← translate[j].s;
14         i ← i + 1;
15     else
16         j ← j + 1;
```

*9.3 Space-efficient construction of the BWT

Perhaps the easiest way to compute the Burrows–Wheeler transform of a string $T\#$ of length $n+1$, where $T[i] \in [1..\sigma]$ for all $i \in [1..n]$ and $0 = \# \notin [1..\sigma]$, consists in building the suffix array of $T\#$ and in scanning the suffix array sequentially, setting $\mathsf{BWT}_{T\#}[i] = T[\mathsf{SA}_{T\#}[i] - 1]$ if $\mathsf{SA}_{T\#}[i] > 1$, and $\mathsf{BWT}_{T\#}[i] = \#$ otherwise. We could use the algorithm described in Theorem 8.12 on page 140 for building $\mathsf{SA}_{T\#}$, spending $O(\sigma + n)$ time and $O(n \log n)$ bits of space. Since $\mathsf{BWT}_{T\#}$ takes $(n+1)\log\sigma$ bits of space, this algorithm would take asymptotically the same space as its output if

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

$\log\sigma \in \Theta(\log n)$. When this is not the case, it is natural to ask for a more space-efficient construction algorithm, and such an algorithm is developed next.

In this section we will repeatedly partition T into *blocks* of equal size B: this is a key strategy for designing efficient algorithms on strings. For convenience, we will work with a version of T whose length is a multiple of B, by appending to the end of T the smallest number of copies of character # such that the length of the resulting padded string is a multiple of B, and such that the padded string contains at least one occurrence of #. Recall that $B \cdot \lceil x/B \rceil$ is the smallest multiple of B that is at least x. Thus, we append $n' - n$ copies of character # to T, where $n' = B \cdot \lceil (n+1)/B \rceil$. To simplify the notation, we call the resulting string X, and we use n' throughout this section to denote its length.

We will interpret a partitioning of X into blocks as a new string X_B of length n'/B, defined on the alphabet $[1..(\sigma+1)^B]$ of all strings of length B on the alphabet $[0..\sigma]$: the "characters" of X_B correspond to the blocks of X. In other words, $X_B[i] = X[(i-1)B+1..iB]$. We assume B to be even, and we denote by `left` (respectively, `right`) the function from $[1..(\sigma+1)^B]$ to $[1..(\sigma+1)^{B/2}]$, such that $\texttt{left}(W)$ returns the first (respectively, second) half of block W. In other words, if $W = w_1 \cdots w_B$, $\texttt{left}(W) = w_1 \cdots w_{B/2}$ and $\texttt{right}(W) = w_{B/2+1} \cdots w_B$.

We will also work with circular rotations of X: specifically, we will denote by $\overleftarrow{X}$ the string $X[B/2+1..n'] \cdot X[1..B/2]$, that is, the string X *circularly rotated to the left by* $B/2$ *positions*, and we will denote by $\overleftarrow{X}_B$ the string on the alphabet $[1..(\sigma+1)^B]$ induced by partitioning $\overleftarrow{X}$ into blocks (see Figure 9.4).

Note that the suffix that starts at position i in $\overleftarrow{X}_B$ equals the half block $P_i = X[B/2+(i-1)B+1..iB]$, followed by the string $S_i = F_{i+1} \cdot X[1..B/2]$, where F_{i+1} is the suffix of X_B that starts at position $i+1$ in X_B, if any (see Figure 9.4). Therefore the order of the suffixes of $\overleftarrow{X}_B$ can be obtained by sorting lexicographically all tuples (P_i, S_i) for $i \in [1..n'/B]$ by their first component and then by their second component. The lexicographic order of the second components coincides with the lexicographic order of the suffixes of X_B for all $i < n'/B$, while the second component of the tuple $(P_{n'/B}, S_{n'/B})$ is not used for determining the order of its tuple. It is thus not surprising that we can derive the BWT of string $\overleftarrow{X}_B$ from the BWT of string X_B.

LEMMA 9.9 *The BWT of string* $\overleftarrow{X}_B$ *can be derived from the BWT of string* X_B *in* $O(n'/B)$ *time and* $O(\sigma^B \cdot \log(n'/B))$ *bits of working space, where* $n' = |X|$.

Proof We use a method that is similar in spirit to the string sorting algorithm described in Lemma 8.7 on page 135. Recall that a character of X_B corresponds to a block of length B of X. First, we build the array C that contains the starting position of the interval of every character of X_B in the BWT of X_B. Specifically, we initialize an array $C'[1..(\sigma+1)^B]$ to zeros, and then we scan the BWT of X_B: for every $i \in [1..n'/B]$ we increment $C'[\mathsf{BWT}_{X_B}[i]]$ by one. Then, we derive the array of starting positions $C[1..(\sigma+1)^B]$ such that $C[c] = \sum_{i=1}^{c-1} C'[i] + 1$ by a linear scan of C'. We also build a similar array D, which contains the starting position of the interval of every *half block* in the BWT of string $\overleftarrow{X}_B$. Specifically, we build the count array $D'[1..(\sigma+1)^{B/2}]$ by scanning the BWT of X_B and by incrementing $D'[\texttt{right}(\mathsf{BWT}_{X_B}[i])]$, that is, we count

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

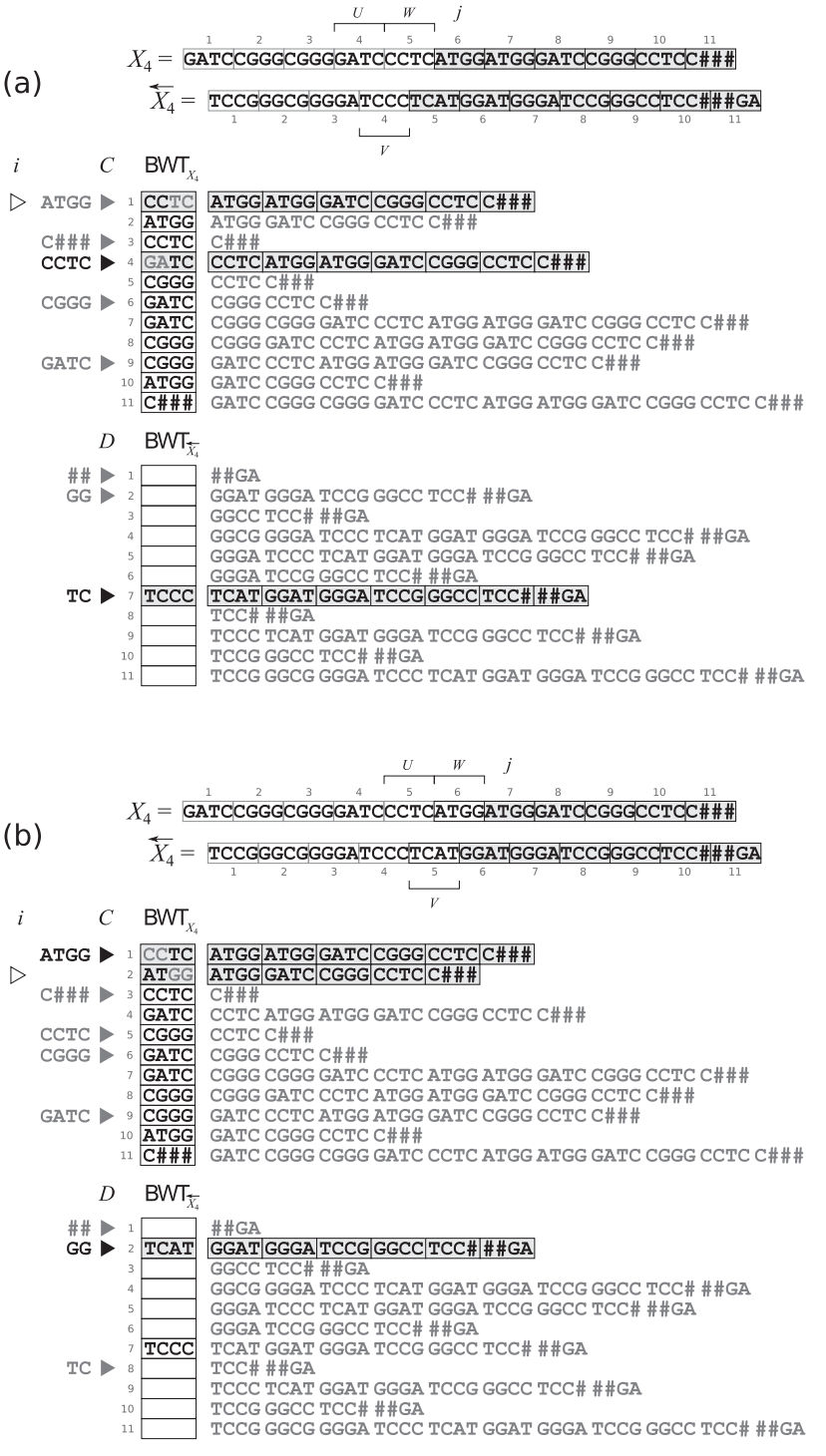

Figure 9.4 The first step (a) and the second step (b) of Lemma 9.9 on a sample string X. The arrays of pointers C and D are represented as labeled gray triangles. The current iteration i is represented as a white triangle. The notation follows Lemma 9.9.

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

just the right half of every character of X_B, for every $i \in [1..n'/B]$. Then, we build the array $D[1..(\sigma+1)^{B/2}]$ such that $D[c] = \sum_{i=1}^{c-1} D'[i] + 1$ by a linear scan of D'.

To build the BWT of $\overleftarrow{X}_B$, it suffices to scan the BWT of X_B from left to right, in a manner similar to Lemma 8.7. Specifically, consider the suffix of X_B at lexicographic rank one, and assume that it starts from position j in X_B (see Figure 9.4(a)). Since the suffix $X_B[j..n'/B]$ of X_B is preceded by the string $W = \mathsf{BWT}_{X_B}[1]$, it follows that the position of suffix $\overleftarrow{X}_B[j-1..n'/B]$ of $\overleftarrow{X}_B$ in the BWT of $\overleftarrow{X}_B$ belongs to the interval that starts at position $D[\mathtt{right}(W)]$: we can thus write string $V = \overleftarrow{X}_B[j-2]$ at this position in the BWT of $\overleftarrow{X}_B$, and increment the pointer $D[\mathtt{right}(W)]$ by one for the following iterations. Clearly $V = \mathtt{right}(U) \cdot \mathtt{left}(W)$, where $U = X_B[j-2]$, so we just need to compute string U.

Recall that $C[W]$ is the starting position of the interval of character W in the BWT of X_B. Since we are scanning the BWT of X_B from left to right, we know that the suffix S that corresponds to the current position i in BWT_{X_B} is the kth suffix that is preceded by W in BWT_{X_B}, where $k = \mathtt{rank}_W(\mathsf{BWT}_{X_B}, 1, i)$. It follows that the position of suffix WS in BWT_{X_B} equals the current value of $C[W]$, therefore we can set $U = \mathsf{BWT}_{X_B}[C[W]]$ and we can increment pointer $C[W]$ by one for the following iterations (see Figure 9.4).

All these operations take $O(n'/B)$ time and $O(\sigma^B \cdot \log(n'/B))$ bits of space. □

The second key observation that we will exploit for building the BWT of X is the fact that the suffixes of $X_{B/2}$ which start at odd positions coincide with the suffixes of X_B, and the suffixes of $X_{B/2}$ that start at even positions coincide with the suffixes of $\overleftarrow{X}_B$. We can thus reconstruct the BWT of $X_{B/2}$ from the BWT of X_B and of $\overleftarrow{X}_B$.

LEMMA 9.10 *Assume that we can read in constant time a block of size B. Then, the BWT of string $X_{B/2}$ can be derived from the BWT of string X_B and from the BWT of string $\overleftarrow{X}_B$, in $O(n' \log \sigma)$ time and $O(\sigma^B \log(n'/B))$ bits of working space, where $n' = |X|$.*

Proof We assume that both of the Burrows–Wheeler transforms provided in the input are represented using wavelet trees (see Section 3.3). The lexicographic rank of a suffix of $X_{B/2}$ equals its rank r with respect to the suffixes of X_B, plus its rank r' with respect to the suffixes of $\overleftarrow{X}_B$, minus one. Thus, we can backward-search string X_B in BWT_{X_B} and in $\mathsf{BWT}_{\overleftarrow{X}_B}$, reading each character of X_B in constant time, and we can determine the ranks r and r' of every suffix of X_B in both transforms. We can thus set $\mathsf{BWT}_{T_{B/2}}[r + r' - 1] = \mathtt{right}(\mathsf{BWT}_{X_B}[r])$. Similarly, we can backward-search $\overleftarrow{X}_B$ in BWT_{X_B} and in $\mathsf{BWT}_{\overleftarrow{X}_B}$, setting $\mathsf{BWT}_{T_{B/2}}[r + r' - 1] = \mathtt{right}(\mathsf{BWT}_{\overleftarrow{X}_B}[r'])$. These searches take in total $O((n'/B) \cdot \log(\sigma^B)) = O(n' \log \sigma)$ time, and they require arrays $C[1..(\sigma+1)^B]$ and $C'[1..(\sigma+1)^B]$ that point to the starting position of the interval of every character in the BWT of X_B and in that of $\overleftarrow{X}_B$, respectively. □

Lemmas 9.9 and 9.10 suggest that one should build the BWT of X in $O(\log B)$ steps: at every step i we compute the BWT of string $X_{B/2^i}$, stopping when $B/2^i = 1$ (see Figure 9.5). To start the algorithm we need the BWT of string X_B for some initial block size B: the most natural way to obtain it is by using a linear-time suffix array construction algorithm that works in $O(\sigma^B + n'/B)$ time and in $O((n'/B)\log(n'/B))$

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

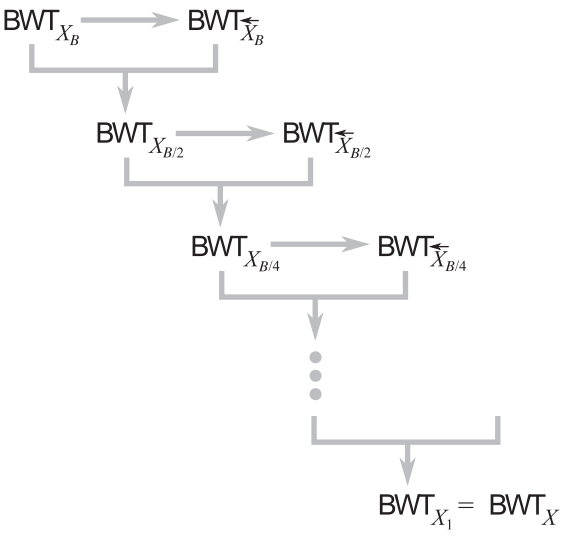

Figure 9.5 High-level structure of the algorithm for building the Burrows–Wheeler transform of a string. The notation follows Section 9.3. Horizontal arrows: Lemma 9.9. Vertical arrows: Lemma 9.10.

bits of space, for example the one described in Section 8.2. We want this first phase to take $O(n')$ time and $O(n' \log \sigma)$ bits of space, or in other words we want to satisfy the following constraints:

i. $\sigma^B \in O(n')$ and
ii. $(n'/B)\log(n'/B) \in O(n' \log \sigma)$, or more strictly $(n'/B)\log n' \in O(n' \log \sigma)$.

The time for every step is $O((n'/B)\log(\sigma^B)) = O(n' \log \sigma)$ for any choice of B, since it is dominated by the time taken by Lemma 9.10. The Burrows–Wheeler transforms of $X_{B/2^i}$ and of $\overleftarrow{X}_{B/2^i}$ at every step i take $O(n' \log \sigma)$ bits of space overall. We want the supporting arrays of pointers used by Lemmas 9.9 and 9.10 to take $O(n')$ bits of space overall, or in other words we want to satisfy the following constraint:

iii. $\sigma^B \log(n'/B) \in O(n')$, or even more strictly $\sigma^B \log n' \in O(n')$.

For simplicity, we also want B to be a power of two. Recall that $2^{\lceil \log x \rceil}$ is the smallest power of two that is at least x. Assume thus that we set $B = 2^{\lceil \log(\log n'/(c \log \sigma)) \rceil}$ for some constant c. Since $\lceil x \rceil \geq x$ for any x, we have that $B \geq \log n'/(c \log \sigma)$, thus constraint (ii) is satisfied by any choice of c. Since $\lceil x \rceil < x+1$, we have that $B < (2/c)\log n'/\log \sigma$, thus constraints (i) and (iii) are satisfied for any $c > 2$. For this choice of B the number of steps in the algorithm becomes $O(\log \log n')$, and we can read a block of size B in constant time as required by Lemma 9.10 since the machine word is assumed to be $\Omega(\log n')$ (see Section 2.2). We have thus proved that the BWT of X can be built in $O(n' \log \sigma \log \log n')$ time and in $O(n' \log \sigma)$ bits of space.

To complete the construction, recall that string X is the original input string T, padded at the end with $k = n' - n \geq 1$ copies of character #. We thus need to derive the

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

BWT of string $T\#$ from the BWT of X, or in other words we need to remove all the occurrences of character # that we added to T, except one. To do so, it suffices to note that $\mathsf{BWT}_X = T[n] \cdot \#^{k-1} \cdot W$ and $\mathsf{BWT}_{T\#} = T[n] \cdot W$, where string W is a permutation of $T[1..n-1]$: see Exercise 9.4. We are thus able to prove the key result of this section.

THEOREM 9.11 *The Burrows–Wheeler transform of a string $T\#$ of length $n+1$, where $T[i] \in [1..\sigma]$ for all $i \in [1..n]$ and $0 = \# \notin [1..\sigma]$, can be built in $O(n \log \sigma \log\log n)$ time and in $O(n \log \sigma)$ bits of space.*

9.4 Bidirectional BWT index

Given a string $T = t_1 t_2 \cdots t_n$ on the alphabet $[1..\sigma]$, consider two BWT indexes, one built on $T\#$ and one built on $\underline{T}\# = t_n t_{n-1} \cdots t_1\#$. Let $\mathbb{I}(W, T)$ be the function that returns the interval in $\mathsf{BWT}_{T\#}$ of the suffixes of $T\#$ that are prefixed by string $W \in [1..\sigma]^+$. Note that the interval $\mathbb{I}(W, T)$ in the *suffix array* of $T\#$ contains all the starting positions of string W in T. Symmetrically, the interval $\mathbb{I}(\underline{W}, \underline{T})$ in the suffix array of $\underline{T}\#$ contains all those positions i such that $n-i+1$ is an *ending position* of string W in T (see Figure 9.6 for an example). In this section we are interested in the following data structure.

DEFINITION 9.12 *Given a string $T \in [1..\sigma]^n$, a* bidirectional BWT index *on T is a data structure that supports the following operations:*

`isLeftMaximal`(i,j)	*Returns* 1 *if substring* $\mathsf{BWT}_{T\#}[i..j]$ *contains at least two distinct characters, and* 0 *otherwise.*
`isRightMaximal`(i,j)	*Returns* 1 *if substring* $\mathsf{BWT}_{\underline{T}\#}[i..j]$ *contains at least two distinct characters, and* 0 *otherwise.*
`enumerateLeft`(i,j)	*Returns all the distinct characters that appear in substring* $\mathsf{BWT}_{T\#}[i..j]$*, in lexicographic order.*
`enumerateRight`(i,j)	*Returns all the distinct characters that appear in* $\mathsf{BWT}_{\underline{T}\#}[i..j]$*, in lexicographic order.*
`extendLeft` $\big(c, \mathbb{I}(W,T), \mathbb{I}(\underline{W},\underline{T})\big)$	*Returns the pair* $\big(\mathbb{I}(cW,T), \mathbb{I}(\underline{Wc},\underline{T})\big)$ *for* $c \in [0..\sigma]$.
`extendRight` $\big(c, \mathbb{I}(W,T), \mathbb{I}(\underline{W},\underline{T})\big)$	*Returns* $\big(\mathbb{I}(Wc,T), \mathbb{I}(c\underline{W},\underline{T})\big)$ *for* $c \in [0..\sigma]$.

For completeness, we introduce here two specializations of the bidirectional BWT index, which will be used extensively in the following chapters.

DEFINITION 9.13 *Given a string $T \in [1..\sigma]^n$, a* forward BWT index *on T is a data structure that supports the following operations:* `isLeftMaximal`(i,j), `enumerateLeft`(i,j), *and* `extendLeft` (c,i,j), *where $[i..j]$ is an interval in* $\mathsf{BWT}_{T\#}$ *and $c \in [0..\sigma]$. Symmetrically, a* reverse BWT index *on T is a data structure that supports the following operations:* `isRightMaximal`(i,j), `enumerateRight`(i,j), *and* `extendRight` (c,i,j), *where $[i..j]$ is an interval in* $\mathsf{BWT}_{\underline{T}\#}$ *and $c \in [0..\sigma]$.*

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

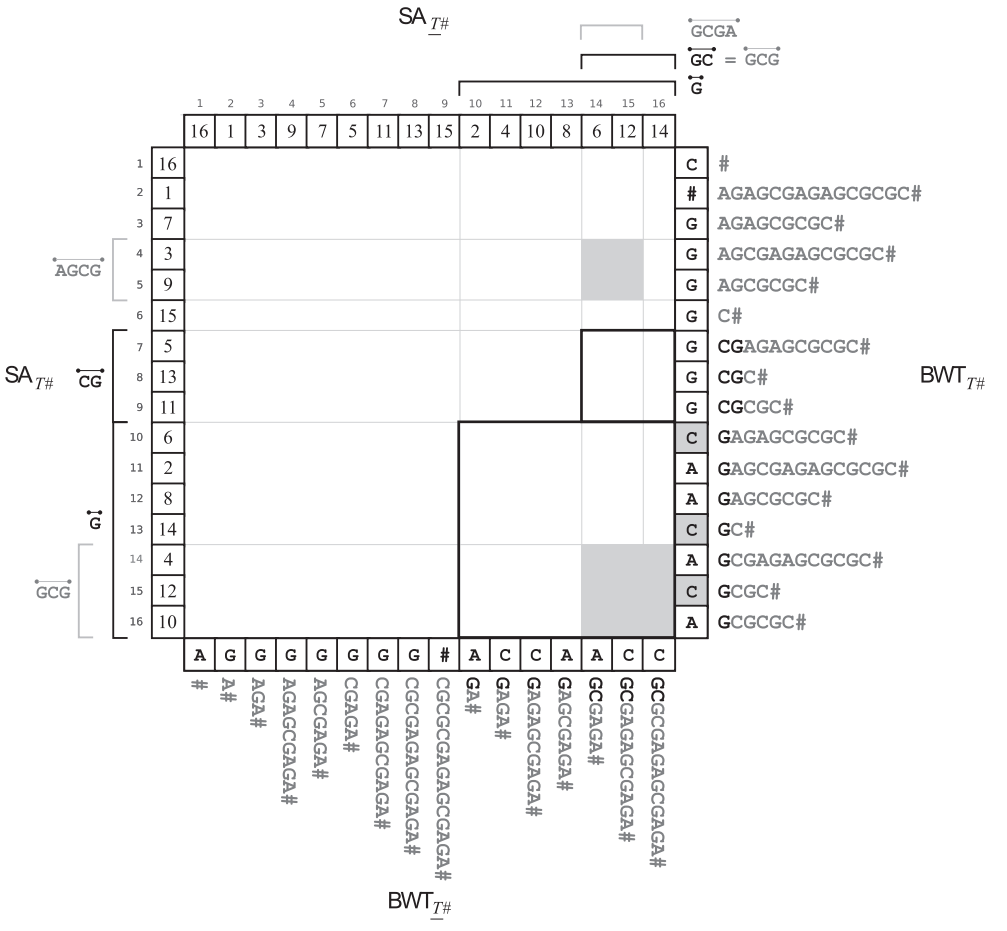

Figure 9.6 Visualizing `extendLeft` in the bidirectional BWT index of string $T =$ `AGAGCGAGAGCGCGC`. The BWT of string $T\#$ is shown on the rows, from top to bottom, and the BWT of string $\underline{T}\# =$ `CGCGCGAGAGCGAGA#` is shown on the columns, from left to right. Squares represent pairs of intervals $(\mathbb{I}(W,T), \mathbb{I}(\underline{W},\underline{T}))$. Consider string `G` (the largest empty square), with interval [10..16] both in $\mathsf{BWT}_{T\#}$ and in $\mathsf{BWT}_{\underline{T}\#}$. A backward step in $\mathsf{BWT}_{T\#}$ with character `C` leads to interval [7..9] (the small empty square). Extending `G` to the left with character `C` in T corresponds to extending `G` to the right with character `C` in $\underline{T}$, therefore the interval of `GC` in $\mathsf{BWT}_{\underline{T}\#}$ is contained in the interval of `G`. Since there are four characters lexicographically smaller than `C` in the interval of `G` in $\mathsf{BWT}_{T\#}$, the interval of `GC` in $\mathsf{BWT}_{\underline{T}\#}$ must start four positions after the beginning of the interval of `G`. Gray squares show the intervals in $\mathsf{BWT}_{T\#}$ and in $\mathsf{BWT}_{\underline{T}\#}$ during the following steps of the backward search for pattern $P =$ `AGCG`. The numbers that correspond to the rows (respectively, columns) of a square are the starting (respectively, ending) positions in T of the substring associated with the square.

Clearly a forward BWT index can be implemented with a BWT index of T, and a reverse BWT index can be implemented with a BWT index of $\underline{T}$. In the bidirectional BWT index, `extendLeft` and `extendRight` are analogous to a standard backward step in $\mathsf{BWT}_{T\#}$ or $\mathsf{BWT}_{\underline{T}\#}$, but they keep the interval of a string W in one BWT

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

synchronized with the interval of its reverse $\underline{W}$ in the other BWT. The following theorem describes how to achieve such synchronization, and Figure 9.6 gives a concrete example.

THEOREM 9.14 *Given a string $T \in [1..\sigma]^n$, there is a representation of the bidirectional BWT index of T that uses $2n \log \sigma (1 + o(1))$ bits of space, supports* `isLeftMaximal`, `isRightMaximal`, `extendLeft`, *and* `extendRight` *in $O(\log \sigma)$ time, and supports* `enumerateLeft` *and* `enumerateRight` *in $O(d \log(\sigma/d))$ time, where d is the size of the output of such operations.*

Proof We build a wavelet tree on $\mathsf{BWT}_{T\#}$ and $\mathsf{BWT}_{\underline{T}\#}$ (see Section 3.3), and we answer $\texttt{isLeftMaximal}(i,j) = 1$ if and only if $\texttt{isRangeUnary}(\mathsf{BWT}_{T\#}, i, j)$ returns false and $\texttt{isRightMaximal}(i,j) = 1$ if and only if $\texttt{isRangeUnary}(\mathsf{BWT}_{\underline{T}}, i, j)$ returns false. We implement $\texttt{enumerateLeft}(i,j)$ by $\texttt{rangeList}(\mathsf{BWT}_{T\#}, i, j, 0, \sigma)$, and we implement $\texttt{enumerateRight}(i,j)$ by $\texttt{rangeList}(\mathsf{BWT}_{\underline{T}\#}, i, j, 0, \sigma)$. The running times required by this theorem are matched by Corollary 3.5.

Consider now $\texttt{extendLeft}\big(c, \mathbb{I}(W,T), \mathbb{I}(\underline{W},\underline{T})\big)$. For simplicity, let $\mathbb{I}(W,T) = [i..j]$ and let $\mathbb{I}(\underline{W},\underline{T}) = [p..q]$. We use $[i'..j']$ and $[p'..q']$ to denote the corresponding intervals of cW and $\underline{W}c$, respectively. As in the standard succinct suffix array, we can get $[i'..j']$ in $O(\log \sigma)$ time using a backward step from interval $[i..j]$, implemented with $\texttt{rank}_c$ queries on the wavelet tree of $\mathsf{BWT}_{T\#}$.

Interval $[p..q]$ in $\mathsf{BWT}_{\underline{T}\#}$ contains all the suffixes of $\underline{T}\#$ that start with $\underline{W}$, or equivalently all the prefixes of $T\#$ that end with W: since $[p'..q']$ contains all the prefixes of $T\#$ that end with cW, it must be contained in $[p..q]$. The number k of prefixes of $T\#$ that end with a string aW with $a < c$ is given by $k = \texttt{rangeCount}(\mathsf{BWT}_{T\#}, i, j, 0, c-1)$, therefore $p' = p + k$. The size of interval $[p'..q']$ equals the size of interval $[i'..j']$, since such intervals correspond to the starting and ending positions of string cW in $T\#$: it follows that $q' = p' + j' - i'$.

The function `extendRight` can be implemented symmetrically. □

The function `extendLeft` allows one to know the interval of every suffix of a string W in $\mathsf{BWT}_{\underline{T}\#}$ while backward-searching W in $\mathsf{BWT}_{T\#}$: see again Figure 9.6 for an example of a backward search in the bidirectional BWT index. More importantly, the bidirectional BWT index allows one to traverse all nodes of the suffix tree of T, using Algorithm 9.3.

THEOREM 9.15 *Algorithm 9.3, executed on the bidirectional BWT index of a string $T \in [1..\sigma]^n$ represented as in Theorem 9.14, outputs all the intervals of the suffix array of $T\#$ that correspond to internal nodes of the suffix tree of T, as well as the length of the label of such nodes, in $O(n \log \sigma)$ time and in $O(\sigma \log^2 n)$ bits of working space.*

Proof Assume by induction that $[i..j]$ is the interval of some internal node v of ST_T at the beginning of the while loop in Algorithm 9.3, thus Σ' contains all characters a such that $a\ell(v)$ is a suffix of $T\#$. There is an internal node w in ST_T such that $\ell(w) = a\ell(v)$ if and only if there is a suffix of $T\#$ that starts with $a\ell(v)b$ and another suffix of $T\#$ that starts with $a\ell(v)c$, where $b \neq c$ are distinct characters. This condition can be tested by checking whether $\texttt{isRightMaximal}(i',j')$ equals one, where $[i'..j'] = \mathbb{I}(\underline{\ell(v)}a, \underline{T})$.

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

Algorithm 9.3: Visiting all the internal nodes of the suffix tree of a string T of length n using the bidirectional BWT index of T

Input: Bidirectional BWT index `idx` of string T, and interval $[1..n+1]$ corresponding to the root of the suffix tree of T

Output: Pairs $(\mathfrak{v}, |\ell(v)|)$ for all internal nodes v of the suffix tree of T, where $\mathfrak{v}$ is the interval of v in the suffix array of $T\#$.

```
1  S ← empty stack;
2  S.push(([1..n+1], [1..n+1], 0));
3  while not S.isEmpty() do
4      ([i..j], [i'..j'], d) ← S.pop();
5      Output ([i..j], d);
6      Σ' ← idx.enumerateLeft(i, j);
7      I ← ∅;
8      for c ∈ Σ' do
9          I ← I ∪ {idx.extendLeft(c, [i..j], [i'..j'])};
10     x ← argmax {j − i : ([i..j], [i'..j']) ∈ I};
11     I ← I \ {x};
12     ([i..j], [i'..j']) ← x;
13     if idx.isRightMaximal(i', j') then
14         S.push(x, d + 1);
15     for ([i..j], [i'..j']) ∈ I do
16         if idx.isRightMaximal(i', j') then
17             S.push(([i..j], [i'..j'], d + 1));
```

Thus Algorithm 9.3 pushes to stack S all and only the intervals that correspond to internal nodes in the suffix tree of T *that can be reached from node v via explicit Weiner links* (or, equivalently, via inverse suffix links). Recall from Section 8.3 that connecting all the internal nodes of the suffix tree of T with explicit Weiner links yields the *suffix-link tree* of T. Thus, Algorithm 9.3 implements a traversal of the suffix-link tree of T, which implies that it visits all the internal nodes of ST_T.

Cases in which `isRightMaximal`$(i', j') = 0$, where $[i'..j']$ is the interval in $\mathsf{BWT}_{\underline{T}\#}$ of a string $a\ell(v)$, correspond to *implicit Weiner links* in ST_T: recall from Observation 8.14 that the total number of such links is $O(n)$. Note also that extending such a string $a\ell(v)$ to the left with additional characters cannot lead to an internal node of ST_T. It follows that Algorithm 9.3 runs in $O(n \log \sigma)$ time, assuming that the bidirectional BWT index of T is implemented as described in Theorem 9.14.

Note that the algorithm considers to push first to the stack the largest interval $[i..j]$. Every other interval is necessarily at most half of the original interval, thus stack S contains at any time intervals from $O(\log n)$ suffix-link tree levels. Every such level contains at most σ interval pairs, each taking $O(\log n)$ bits. □

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

In practice it is more efficient to combine the implementations of `enumerateLeft` and of `extendLeft` (and symmetrically of `enumerateRight` and of `extendRight`) in Algorithm 9.3: see Exercise 9.7. Note that Algorithm 9.3 can be used to implement in $O(n \log \sigma)$ time and in $2n \log \sigma (1 + o(1)) + O(\sigma \log^2 n)$ bits of space *any algorithm* that

- traverses all nodes of ST_T;
- performs a number of additional operations which is linear in n or in the size of the output;
- does not depend on the *order* in which the nodes of the suffix tree are visited.

We will see in later chapters that a number of fundamental sequence analysis tasks, such as detecting maximal repeats and maximal unique matches, satisfy such properties. Some of the applications we will describe visit only nodes of the suffix tree whose label is of length at most k, where k is a constant: in such cases the size of the stack is $O(n \log \sigma)$ even without using the trick of Algorithm 9.3, and we can traverse the suffix-link tree of T depth-first.

*9.4.1 Visiting all nodes of the suffix tree with just one BWT

In order to implement Algorithm 9.3 we do not need all the expressive power of the bidirectional BWT index of T. Specifically, we do not need operations `enumerateRight` and `extendRight`, but we do need `isRightMaximal`: we can thus replace the BWT of $\underline{T}\#$ with a bitvector `runs`$[1..n+1]$ such that `runs`$[i] = 1$ if and only if $i > 1$ and $\mathsf{BWT}_{\underline{T}\#}[i] \neq \mathsf{BWT}_{\underline{T}\#}[i-1]$. We can build this vector in a linear scan of $\mathsf{BWT}_{\underline{T}\#}$, and we index it to support `rank` queries in constant time. We can then implement `isRightMaximal`(i,j) by checking whether there is a one in `runs`$[i+1..j]$, or equivalently by checking whether $\texttt{rank}_1(\texttt{runs}, j) - \texttt{rank}_1(\texttt{runs}, i) \geq 1$. In this section we show how to implement Algorithm 9.3 without even using `runs`, or equivalently *using just* $\mathsf{BWT}_{T\#}$, without any penalty in running time.

Let v be a node of ST_T, and let $\gamma(v,c)$ be the number of children of v that have a Weiner link (implicit or explicit) labeled by character $c \in [1..\sigma]$. The following property, visualized in Figure 9.7, is key for using just one BWT in Algorithm 9.3. Its easy proof is left to Exercise 9.6.

> PROPERTY 1 *There is an explicit Weiner link (v,w) labeled by character c in ST_T if and only if $\gamma(v,c) \geq 2$. In this case, node w has exactly $\gamma(v,c)$ children. If $\gamma(v,c) = 1$, then the Weiner link from v labeled by c is implicit. If $\gamma(v,c) = 0$, then no Weiner link from v is labeled by c.*

Thus, if we maintain at every step of Algorithm 9.3 the intervals of the children of a node v of ST_T in addition to the interval of v itself, we could generate all Weiner links

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

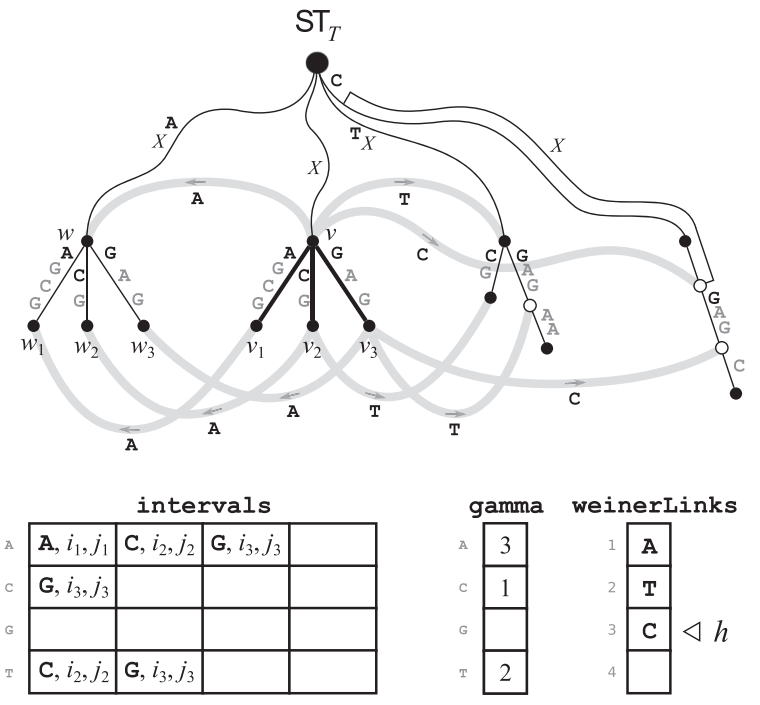

Figure 9.7 Theorem 9.16 applied to node v. Gray directed arcs represent implicit and explicit Weiner links. White dots represent the destinations of implicit Weiner links. Among all strings prefixed by string $\ell(v) = X$, only those prefixed by XGAG are preceded by C. It follows that CX is always followed by G and it is not a node in the suffix tree of T, thus the Weiner link from v labeled by C is implicit. Conversely, XAGCG, XCG, and XGAG are preceded by an A, so AX is a node in the suffix tree, and the Weiner link from v labeled by A is explicit.

from v and check whether they are explicit. The following theorem makes this intuition more precise.

THEOREM 9.16 *There is an algorithm that, given the BWT index of a string T, outputs all the intervals of the suffix array of $T\#$ that correspond to internal nodes of the suffix tree of T, as well as the length of the label of such nodes, in $O(n \log \sigma)$ time and $O(\sigma^2 \log^2 n)$ bits of working space.*

Proof As anticipated, we adapt Algorithm 9.3 to use just $\mathsf{BWT}_{T\#}$ rather than the full bidirectional BWT index of T. Let v be an internal node of the suffix tree of T, let $v_1, v_2, \ldots, v_k$ be its children in the suffix tree in lexicographic order, and let $\phi(v_i)$ be the first character of the label of arc (v, v_i). Without loss of generality, we consider just characters in $[1..\sigma]$ in what follows. Assume that we know both the interval $\overleftrightarrow{v}$ of v in $\mathsf{BWT}_{T\#}$, and the interval $\overleftrightarrow{v_i}$ of every child v_i.

Let `weinerLinks`$[1..\sigma]$ be a vector of characters, and let h be the number of non-zero characters in this vector. We will store in vector `weinerLinks` the labels of all (implicit and explicit) Weiner links that start from v. Let `intervals`$[1..\sigma][1..\sigma]$ be a matrix, whose rows correspond to all possible labels of a Weiner link from v. We

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

will store in row c of matrix `intervals` a set of triplets $(\phi(v_i), \overbrace{c\ell(v_i)}, \overbrace{c\ell(v_i)})$, where intervals refer to $\mathsf{BWT}_{T\#}$. By encoding $\phi(v_i)$, every such triplet identifies a child v_i of v in the suffix tree, and it specifies the interval of the destination of a Weiner link from v_i labeled by c. We use array `gamma`$[1..\sigma]$ to maintain the number of children of v that have a Weiner link labeled by c, for every $c \in [1..\sigma]$. In other words, `gamma`$[c] = \gamma(v, c)$. See Figure 9.7 for an example of such arrays.

For every child v_i of v, we enumerate all the distinct characters that occur in $\mathsf{BWT}_{T\#}[\overbrace{v_i}]$: for every such character c, we compute $\overbrace{c\ell(v_i)}$, we increment counter `gamma`$[c]$ by one, and we store triplet $(\phi(v, i), \overbrace{c\ell(v_i)}, \overbrace{c\ell(v_i)})$ in `intervals`$[c]$ [`gamma`$[c]$]. If `gamma`$[c]$ transitioned from zero to one, we increment h by one and we set `weinerLinks`$[h] = c$. At the end of this process, the labels of all Weiner links that start from v are stored in `weinerLinks` (not necessarily in lexicographic order), and by Property 1 we can detect whether the Weiner link from v labeled by character `weinerLinks`$[i]$ is explicit by checking whether `gamma`[`weinerLinks`$[i]] > 1$.

Let w be the destination of the explicit Weiner link from v labeled by character c. Note that the characters that appear in row c of matrix `intervals` are sorted lexicographically, and that the corresponding intervals are exactly the intervals of the children $w_1, w_2, \ldots, w_{\gamma(v,c)}$ of w in the suffix tree of T, in lexicographic order of $\ell(w_i)$. It follows that such intervals are adjacent in $\mathsf{SA}_{T\#}$, thus $\overbrace{w}$ equals the first position of the first interval in row c, and $\overbrace{\overline{w}}$ equals the last position of the last interval in row c. We can then push on the stack used by Algorithm 9.3 intervals $\overbrace{w_1}, \overbrace{w_2}, \ldots, \overbrace{w_{\gamma(v,c)}}$ and number $\gamma(v, c)$. After having processed in the same way all destinations w of explicit Weiner links from v, we can reset h to zero, and `gamma`$[c]$ to zero for all labels c of (implicit or explicit) Weiner links of v.

The algorithm can be initialized with the interval of the root of ST_T and with the interval of the distinct characters in $\mathsf{SA}_{T\#}$, which can be derived using the C array of backward search. Using the same stack trick as in Algorithm 9.3, the stack contains $O(\log n)$ levels of the suffix-link tree of T at any given time. Every such level contains at most σ nodes, and each node is represented by the at most σ intervals of its children in the suffix tree. Every such interval takes $O(\log n)$ bits, therefore the total space required by the stack is $O(\log^2 n \cdot \sigma^2)$. □

Note that the algorithm in Theorem 9.16 builds, for every internal node, its children in the suffix tree, thus it can be used to implement any algorithm that invokes `enumerateRight` and `extendRight` only on the intervals of $\mathsf{BWT}_{T\#}$ associated with internal nodes of the suffix tree. We will see in later chapters that this is the case for a number of fundamental sequence analysis tasks.

*9.5 BWT index for labeled trees

The idea of sorting lexicographically all the suffixes of a string, and of representing the resulting list in small space by storing just the character that precedes every suffix, could be applied to labeled trees: specifically, we could sort in lexicographic order all

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

the *paths* that start from the root, and we could represent such a sorted list in small space by storing just the labels of the *children* of the nodes that correspond to every path. In this section we make this intuition more precise, by describing a representation technique for labeled trees that is flexible enough to generalize in Section 9.6 to labeled DAGs, and to apply almost verbatim in Section 9.7.1 to de Bruijn graphs.

Let $\mathcal{T} = (V, E, \Sigma)$ be a rooted tree with a set of nodes V, a root $r \in V$, and a set of *directed* arcs E, such that every node $v \in V$ is labeled by a character $\ell(v) \in \Sigma = [1..\sigma]$. Without loss of generality, we assume that $\ell(r) = \#$, where $\# = 0 \notin \Sigma$. Arcs are directed *from child to parent* (see Figure 9.8(a)): formally, $(u, v) \in E$ if and only if u is a child of v in $\mathcal{T}$. We also assume that V obeys a *partial order* $\prec$, in the sense that there is a total order among the children of every node. In particular, we extend the partial order $\prec$ to a *total order* $\prec^*$ *on the entire* V, which satisfies the following two properties:

- $u \prec v$ implies $u \prec^* v$;
- $u \prec^* v$ implies $u' \prec^* v'$ for every child u' of u and every child v' of v.

Such a total order can be induced, for example, by traversing $\mathcal{T}$ depth-first in the order imposed by $\prec$, and by printing the children of every traversed node. We denote as a *path* a sequence of nodes $P = v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $i \in [1..k-1]$. We lift the label operator ℓ to paths, by defining $\ell(P) = \ell(v_1) \cdot \ell(v_2) \cdot \cdots \cdot \ell(v_k)$, and we set $\bar{\ell}(v) = \ell(v_1, v_2, \ldots, v_k)$ for $v \in V$, where $v_1, v_2, \ldots, v_k$ is the path that connects $v = v_1$ to the root $v_k = r$. We call $\bar{\ell}(v)$ the *extended label* of node v. Note that, if $\mathcal{T}$ is a chain, $\bar{\ell}(v)$ corresponds to a *suffix* of $\mathcal{T}$. Given a string $W \in [0..\sigma]^*$, we use $\texttt{count}(W)$ to denote the number of paths P in $\mathcal{T}$ such that $\ell(P) = W$. For brevity, we use the shorthands n and m to denote $|V|$ and the number of leaves of $\mathcal{T}$, respectively.

For a node $v \in V$, let $\texttt{last}(v)$ be a binary flag that equals one if and only if v is the last child of its parent according to $\prec$, and let $\texttt{internal}(v)$ be a binary flag that equals one if and only if v is an internal node of $\mathcal{T}$. Assume that for every node v in the order induced by $\prec^*$, and for every arc $(u, v) \in E$, we concatenate to a list L the quadruplet $(\ell(u), \bar{\ell}(v), \texttt{last}(u), \texttt{internal}(u))$ if v is an internal node, and the quadruplet $(\#, \bar{\ell}(v), 1, 1)$ if v is a leaf. The first type of quadruplet represents arc $(u, v) \in E$, and the second type of quadruplet represents an artificial arc $(r, v) \notin E$. Note that the same quadruplet can be generated by different arcs, and that every string W that labels a path of $\mathcal{T}$ is the prefix of the second component of at least $\texttt{count}(W)$ quadruplets.

Assume now that we perform a *stable sort* of L in lexicographic order by the second component of each quadruplet. After such sorting, the incoming arcs of a node still form a consecutive interval of L in the order imposed by $\prec$, and the intervals of nodes v and w with $\bar{\ell}(v) = \bar{\ell}(w)$ still occur in L in the order imposed by $\prec^*$ (see Figure 9.8(b)). We denote by $L[i]$ the ith quadruplet in the sorted L, and by $L[i].\ell$, $L[i].\bar{\ell}$, $L[i].\texttt{last}$, and $L[i].\texttt{internal}$ the corresponding fields of quadruplet $L[i]$. Note that, if $\mathcal{T}$ is a chain, this process is analogous to printing all the suffixes of $\mathcal{T}$, along with their preceding characters. Thus, in the case of a chain, sorting the quadruplets in L by field $\bar{\ell}$ and printing the field ℓ of the sorted list yields precisely the Burrows–Wheeler transform of $\mathcal{T}$.

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

Recall that a suffix corresponds to a unique position in the Burrows–Wheeler transform of a string; analogously, in what follows we identify a node $v \in V$ with *the interval* $\mathbb{I}(v)$ *of* L *that contains all quadruplets* $(\ell(u), \bar{\ell}(v), \texttt{last}(u), \texttt{internal}(u))$, or equivalently all the incoming arcs of v in $\mathcal{T}$. List L can be represented implicitly, by just storing arrays $\texttt{labels}[1..n+m-1]$, $\texttt{internal}[1..n+m-1]$, $\texttt{last}[1..n+m-1]$, and $C[0..\sigma]$, where $\texttt{labels}[i] = L[i].\ell$, $\texttt{internal}[i] = L[i].\texttt{internal}$, $\texttt{last}[i] = L[i].\texttt{last}$, and $C[c]$ contains the number of nodes $v \in V$ whose extended label $\bar{\ell}(v)$ is lexicographically smaller than character $c \in [0..\sigma]$. We call this set of arrays the *Burrows–Wheeler index of tree* $\mathcal{T}$ (see Figure 9.8(b)).

DEFINITION 9.17 *Given a labeled tree* $\mathcal{T}$ *with* n *nodes and* m *leaves, the* Burrows–Wheeler index of $\mathcal{T}$*, denoted by* $\mathsf{BWT}_{\mathcal{T}}$*, is a data structure that consists of*

- *the array of characters* `labels`*, encoded using a wavelet tree (see Section 3.3);*
- *bitvector* `last`*, indexed to support* `rank` *and* `select` *queries in constant time as described in Section 3.2;*
- *bitvector* `internal`*;*
- *the count array* C*.*

See the main text for the definition of such arrays. $\mathsf{BWT}_{\mathcal{T}}$ takes $(n+m)\log\sigma(1+o(1))+2(n+m)+o(n+m)$ bits of space.

*9.5.1 Moving top-down

$\mathsf{BWT}_{\mathcal{T}}$ shares most of the properties of its textual counterpart. In particular, nodes $v \in V$ such that $\bar{\ell}(v)$ starts with the same string $W \in [0..\sigma]^*$ form a contiguous interval $\overrightarrow{W}$ in $\mathsf{BWT}_{\mathcal{T}}$, and all arcs (u, v) with the same label $\ell(u)$ appear in lexicographic order of $\bar{\ell}(v)$ across all the intervals of nodes in $\mathsf{BWT}_{\mathcal{T}}$, with ties broken according to $\prec^*$. As already anticipated, all the $\texttt{inDegree}(v)$ incoming arcs of a node $v \in V$ are contained in its interval $\overrightarrow{v}$, they follow the total order imposed by $\prec$, and the subarray $\texttt{last}[\overrightarrow{v}]$ is a unary encoding of $\texttt{inDegree}(v)$. This immediately allows to compute $\texttt{inDegree}(v) = |\overrightarrow{v}|$ and to determine the number of children of v labeled by a given character $a \in [1..\sigma]$, by computing $\texttt{rank}_a(\texttt{labels}, \overrightarrow{v}) - \texttt{rank}_a(\texttt{labels}, \overleftarrow{v} - 1)$.

$\mathsf{BWT}_{\mathcal{T}}$ shares another key property with the Burrows–Wheeler transform of a string.

PROPERTY 2 *If two nodes* u *and* v *satisfy* $\ell(u) = \ell(v)$*, the order between intervals* $\overrightarrow{u}$ *and* $\overrightarrow{v}$ *is the same as the order between arcs* $(u, \texttt{parent}(u))$ *and* $(v, \texttt{parent}(v))$.

The proof of this property is straightforward and is left to Exercise 9.8. It follows that, if (u, v) is the ith arc in $\mathsf{BWT}_{\mathcal{T}}$ with $\ell(u) = a$, the interval of u is the ith sub-interval of $\overrightarrow{a}$, or equivalently the sub-interval of $\texttt{last}[\overrightarrow{a}]$ delimited by the $(i-1)$th and the ith ones.

Assume thus that we want to move from $\overrightarrow{v}$ to $\overrightarrow{u}$, where u is the kth child of v labeled by character $c \in [1..\sigma]$ according to order $\prec$. Arc (u, v) has lexicographic rank

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

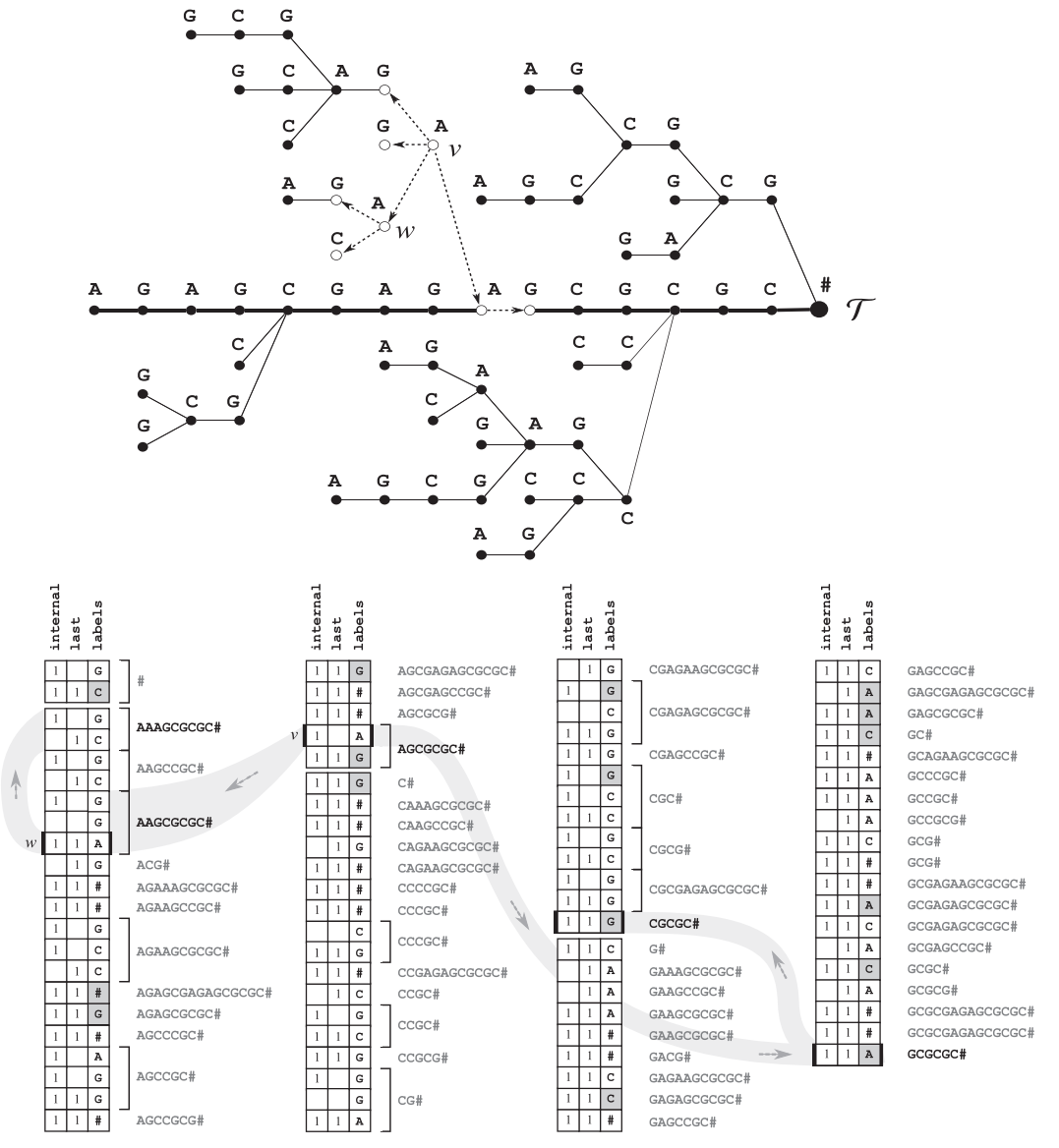

Figure 9.8 A labeled tree $\mathcal{T}$ (top panel) and its Burrows–Wheeler transform $\mathsf{BWT}_{\mathcal{T}}$ built using a pre-order traversal (bottom panel, laid out from top to bottom, from left to right). Arcs in the top panel are assumed to be directed from left to right. For clarity, zeros are omitted from bitvectors in the bottom panel. Spaces in $\mathsf{BWT}_{\mathcal{T}}$ mark the starting position of the interval of every character in $[1..\sigma]$. Black borders to the left and to the right of a row represent $\mathbb{I}'$ intervals. Gray cells correspond to the BWT of string $T = \texttt{AGAGCGAGAGCGCGC\#}$, which labels the thick path in the top panel. See Figure 9.1 for a representation of the BWT and of the suffix array of string T. Consider node v in the top panel, with $\bar{\ell}(v) = \texttt{AAGCGCGC\#}$: there are two occurrences of A before $\mathbb{I}'(v)$ in $\mathsf{BWT}_{\mathcal{T}}$, thus $\mathbb{I}(v)$ is the third block in the interval of character A. Similarly, AGCGCGC# is the 16th sub-interval in lexicographic order inside the interval of character A, thus $\mathbb{I}'(\texttt{parent}(v))$ corresponds to the 16th A in $\mathsf{BWT}_{\mathcal{T}}$. The bottom panel continues the navigation of $\mathcal{T}$ to $\mathbb{I}(w)$ and to $\mathbb{I}'(\texttt{parent}(\texttt{parent}(v)))$ (dashed arrows).

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

$r = \texttt{rank}_c(\texttt{labels}, \overleftarrow{v} - 1) + k$ among all arcs with $\ell(u) = c$, thus $\overleftarrow{u}$ is the rth sub-interval of $\overleftarrow{c}$:

$$\begin{aligned}\texttt{child}(\overleftarrow{v}, c, k) = \Big[&\texttt{select}_1\big(\texttt{last}, \mathsf{LF}(\overleftarrow{v} - 1, c) + k - 1\big) + 1\ ..\\ &\texttt{select}_1(\texttt{last}, \mathsf{LF}(\overleftarrow{v} - 1, c) + k)\ \Big],\\ \mathsf{LF}(i, c) = C[c] &+ \texttt{rank}_c(\texttt{labels}, i),\end{aligned} \tag{9.2}$$

where LF is a generalization of the function described in Section 9.2.1. See Figure 9.8(b) for a concrete example. Note that this operation takes $O(\log \sigma)$ time. Moving to the kth child of v according to order $\prec$ is equally easy, and it is left as an exercise.

Recall that, for a given string $W \in [0..\sigma]^+$, $\overleftarrow{W}$ contains the intervals of all nodes $v \in V$ whose extended label $\bar{\ell}(v)$ is prefixed by W, and for every such node it contains all its incoming arcs in the order imposed by $\prec$. Thus, $\texttt{rank}_1(\texttt{last}, \overleftarrow{W}) - \texttt{rank}_1(\texttt{last}, \overleftarrow{W} - 1)$ is the number of paths (not necessarily ending at the root) labeled by W in $\mathcal{T}$, and $|\overleftarrow{W}|$ is the number of arcs $(u, v) \in E$ such that $\bar{\ell}(u) = cWQ$ for some $c \in [0..\sigma]$ and $Q \in [0..\sigma]^*$. We can thus lift the function $\texttt{child}$ to map $\overleftarrow{W}$ onto $\overleftarrow{cW}$ for any $c \in \texttt{labels}[\overleftarrow{W}]$, as follows:

$$\begin{aligned}\texttt{extendLeft}(c, \overleftarrow{W}, \overleftarrow{W}) = \Big[&\texttt{select}_1\big(\texttt{last}, \mathsf{LF}(\overleftarrow{W} - 1, c)\big) + 1\ ..\\ &\texttt{select}_1\big(\texttt{last}, \mathsf{LF}(\overleftarrow{W}, c)\big)\ \Big].\end{aligned} \tag{9.3}$$

This procedure clearly takes $O(\log \sigma)$ time, and it supports a generalization of the backward search algorithm described in Section 9.2.2 that returns the interval that contains the first nodes of all paths labeled by a string W of length k in $\mathcal{T}$: the search proceeds again from right to left along W, computing iteratively $\overleftarrow{W_{i..k}} = \texttt{extendLeft}\left(W[i], \overleftarrow{W_{i+1..k}}, \overleftarrow{W_{i+1..k}}\right)$. More generally, the following lemma holds.

LEMMA 9.18 $\mathsf{BWT}_\mathcal{T}$ *is a* forward BWT index *(Definition 9.13). The operations* $\texttt{extendLeft}$ *and* $\texttt{isLeftMaximal}$ *can be implemented in* $O(\log \sigma)$ *time, and* $\texttt{enumerateLeft}$ *can be implemented in* $O(d \log(\sigma/d))$ *time, where* d *is the size of the output.*

*9.5.2 Moving bottom-up

Moving from a node u to $v = \texttt{parent}(u)$ requires one to introduce a different representation for nodes. Specifically, we assign to *u the unique position i of arc (u, v) in* L, or equivalently the position of the quadruplet $(\ell(u), \bar{\ell}(v), \texttt{last}(u), \texttt{internal}(u))$. We denote this unary interval by $\mathbb{I}'(u) = i$ to distinguish it from the interval $\mathbb{I}(v)$ of size $\texttt{inDegree}(v)$ used until now. To compute $\texttt{parent}(\mathbb{I}'(u)) = \mathbb{I}'(v)$, we proceed as follows. Let $\bar{\ell}(u) = abW$, where a and b are characters in $[1..\sigma]$ and $W \in [0..\sigma]^*$: we want to move to the position in $\mathsf{BWT}_\mathcal{T}$ of the arc (v, w) such that $\bar{\ell}(v) = bW$. Character b satisfies $C[b] < \texttt{rank}_1(\texttt{last}, i - 1) + 1 \leq C[b + 1]$, which can be determined by a binary search over C, and the lexicographic rank of $\overleftarrow{bW}$ inside $\overleftarrow{b}$ is $\texttt{rank}_1(\texttt{last}, i - 1) - C[b] + 1$, therefore

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

$$\texttt{parent}(i, b) = \texttt{select}_b\big(\texttt{labels}, \texttt{rank}_1(\texttt{last}, i-1) - C[b] + 1\big). \qquad (9.4)$$

See Figure 9.8(b) for a concrete example. This implementation of `parent` clearly takes $O(\log \sigma)$ time.

By Property 2, we can convert $\mathbb{I}'(v)$ to $\mathbb{I}(v)$ by simply computing $\mathbb{I}(v) = \texttt{child}(\mathbb{I}'(v), c)$, where $c = \texttt{labels}[\mathbb{I}'(v)]$. Indeed, if (v, w) is the ith arc in $\mathsf{BWT}_{\mathcal{T}}$ with $\ell(v) = c$, then all arcs (u, v) are clustered in the ith sub-interval of $\overrightarrow{c}$, which is the sub-interval of $\texttt{last}[\overrightarrow{c}]$ delimited by the $(i-1)$th and the ith ones. Converting $\mathbb{I}(v)$ into $\mathbb{I}'(v)$ is equally straightforward. We can thus navigate $\mathcal{T}$ in both directions and in any order using $\mathbb{I}(v)$ and $\mathbb{I}'(v)$. Note that using $\mathbb{I}(v)$ and $\mathbb{I}'(v)$ is equivalent to using as a unique identifier of a node v its rank in the list of all nodes, sorted lexicographically by $\bar{\ell}(v)$ and with ties broken according to the total order $\prec^*$.

*9.5.3 Construction and space complexity

It is easy to see that we can remove all occurrences of # from `labels`, since we are not interested in moving from a leaf to the root, or in moving from the root to its parent. The resulting index takes $2n + n \log \sigma\,(1 + o(1)) + o(n)$ bits of space overall, which matches the information-theoretic lower bound up to lower-order terms. Indeed, the number of distinct, unlabeled, full binary trees with n internal nodes is the *Catalan number* $C_n = \binom{2n}{n}/(n+1)$ (see also Section 6.6.1): thus, we need at least $\log(C_n) \approx 2n - \Theta(\log n)$ bits to encode the topology of $\mathcal{T}$, and we need at least $n \log \sigma$ bits to encode its labels. In contrast, a naive pointer-based implementation of $\mathcal{T}$ would take $O(n \log n + n \log \sigma)$ bits of space, and it would require $O(n|W|)$ time to find or count all paths labeled by $W \in [0..\sigma]^+$.

It is easy to build $\mathsf{BWT}_{\mathcal{T}}$ in $O(n \log n)$ time by extending the prefix-doubling technique used in Lemma 8.8 for building suffix arrays: see Exercise 9.10. The algorithm for filling the partially empty suffix array of a string initialized with the set of its small suffixes (Theorem 8.12) can be applied almost verbatim to $\mathsf{BWT}_{\mathcal{T}}$ as well, but it achieves linear time only when the topology and the labels of $\mathcal{T}$ are coupled: see Exercise 9.11.

*9.6 BWT index for labeled DAGs

As anticipated in Section 9.5, the technique used for representing labeled trees in small space applies with minor adaptations to labeled, directed, acyclic graphs. Genome-scale labeled DAGs are the natural data structure to represent millions of *variants* of a single chromosome, observed across hundreds of thousands of individuals in a population, as paths that share common segments in the same graph: see Figure 9.9(a) for a small example, and Section 10.7.2 for applications of labeled DAGs to read alignment.

Let $G = (V, E, \Sigma)$ be a directed acyclic graph with a set of vertices V, unique source $s \in V$, and unique sink $t \in V$, such that every vertex $v \in V \setminus \{s, t\}$ is labeled by a character $\ell(v)$ on alphabet $\Sigma = [1..\sigma]$. Without loss of generality, we assume that

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

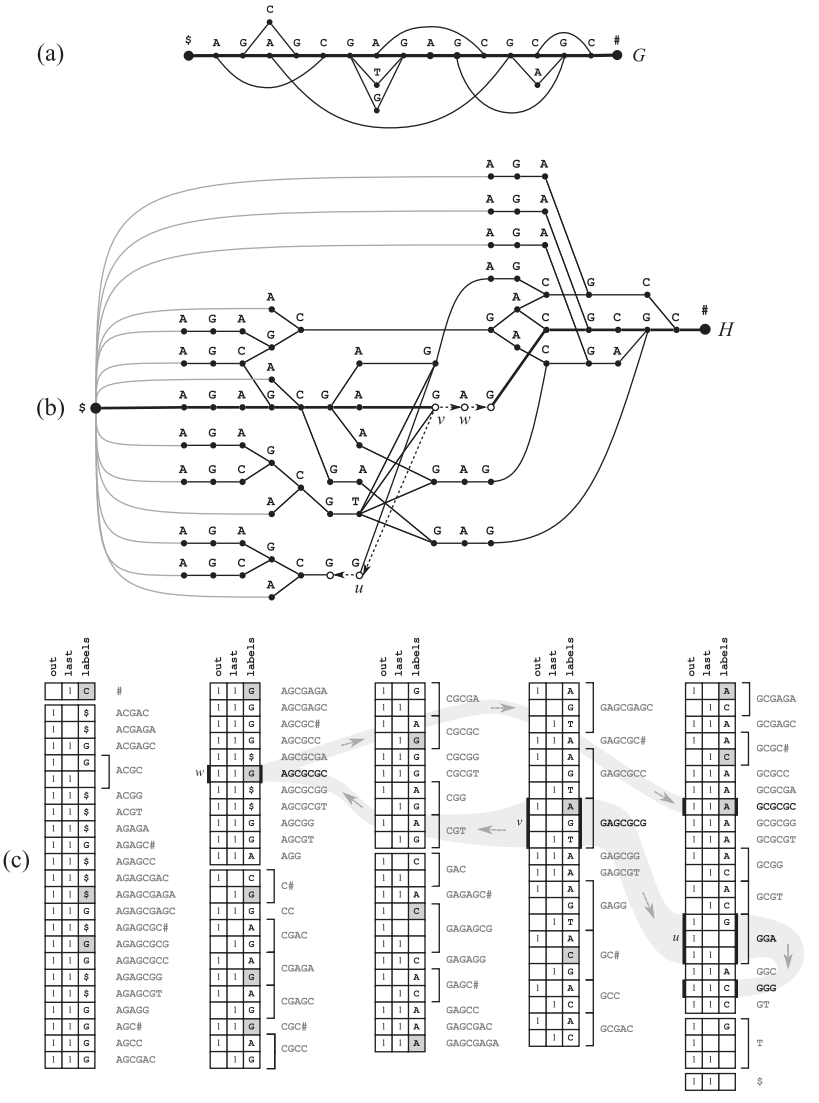

Figure 9.9 (a) A directed, acyclic, labeled, reverse-deterministic graph G, representing string $T = \$\texttt{AGAGCGAGAGCGCGC\#}$ with three single-character variations and five deletions. Arcs are directed from left to right. (b) The equivalent strongly distinguishable DAG H. (c) BWT_H, laid out from top to bottom, from left to right, with the corresponding set of distinguishing prefixes. For clarity, zeros are omitted from bitvectors. Spaces mark the starting position of the interval of every character in $[1..\sigma + 1]$. Black borders to the left and to the right of a row represent intervals that correspond to vertices. Gray cells correspond to the Burrows–Wheeler transform of string $T\#$, which labels the thick path in panels (a) and (b). See Figure 9.1 for a representation of the BWT and of the suffix array of string $T\#$. Consider vertex v in panel (b), corresponding to interval $[i..j]$ in BWT_H, and assume that we want to compute the interval of u by issuing `inNeighbor`$([i..j], \texttt{G})$. There are 32 occurrences of `G` up to position i, and there are $C[\texttt{G}] = 48$ arcs (x, y) with P_x lexicographically smaller than `G`, therefore the interval of u contains the 80th one in `out`. Similarly, assume that we want to move from v to its only out-neighbor w by issuing `outNeighbor`$([i..j], 1)$. There are 62 arcs (x, y) where P_x is lexicographically smaller than P_v, 14 of which have P_x prefixed by $\ell(v) = \texttt{G}$. Therefore, the interval of w contains the 15th `G` in `labels`. Panel (c) continues the navigation of H to the in-neighbor of vertex u and to the out-neighbor of vertex w (dashed arrows in panels (b) and (c)).

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

$\ell(s) = \$$ and $\ell(t) = \#$, where $\# = 0$ and $\$ = \sigma + 1$. As done in Section 9.5, we lift operator ℓ to paths, and we set $\bar{\ell}(v)$ for a vertex $v \in V$ to be the *set of path labels* $\{\ell(v_1, v_2, \ldots, v_k) \mid v_1 = v, v_k = t, (v_i, v_{i+1}) \in E \ \forall\ i \in [1..k-1]\}$. We say that G is *forward-deterministic* if every vertex has at most one out-neighbor with label a for every $a \in [1..\sigma]$. The notion of *reverse-deterministic* is symmetric. We use the shorthand n to denote $|V|$. We say that G is *distinguishable* if, for every pair of vertices v and w in V, $v \neq w$ implies either that P is lexicographically smaller than P' for all $P \in \bar{\ell}(v)$ and $P' \in \bar{\ell}(w)$, or that P is lexicographically larger than P' for all $P \in \bar{\ell}(v)$ and $P' \in \bar{\ell}(w)$. Note that distinguishability implies reverse-determinism. We say that vertex v is *strongly distinguishable* if there is a string $P_v \in [1..\sigma]^+$ such that all strings in $\bar{\ell}(v)$ are prefixed by P_v, and such that $v \neq w$ implies that no string in $\bar{\ell}(w)$ is prefixed by P_v. We call P_v the *distinguishing prefix* of vertex v. Although everything we describe in this section applies to distinguishable DAGs, we assume that every vertex in G is strongly distinguishable to simplify the notation.

Assume that, for every arc $e = (u, v) \in E$, we concatenate to a list L^- the triplet $(\ell(u), P_v, e)$, and assume that we sort L^- in lexicographic order by the second component of each triplet, breaking ties arbitrarily. We denote by $L^-.\ell$ the sequence built by taking the first component of each triplet in the sorted list L^-. Similarly, assume that for every arc $e = (u, v) \in E$, we concatenate to a list L^+ the pair (P_u, e), and that we sort L^+ in lexicographic order by the first component of each pair, breaking ties using the lexicographic order of P_v. In contrast to the case of labeled trees, let array $C[0..\sigma + 1]$ store at position c the number of *arcs* $(u, v) \in E$ with $\ell(u) < c$. Since G is strongly distinguishable, every vertex $v \in V$ can be assigned to the contiguous interval in L^- or L^+ that corresponds to P_v. The key property of L^- and L^+ is that we can easily move from the position of an arc in L^- to its position in L^+.

PROPERTY 3 *Let (u, v) be an arc with $\ell(u) = c$, and assume that it occurs at position i in L^-. The position of (u, v) in L^+ is $\mathsf{LF}(i, c) = C[c] + \mathtt{rank}_c(L^-.\ell, i)$.*

The proof of this property is straightforward: since (u, v) is the jth arc with $\ell(u) = c$ in L^-, where $j = \mathtt{rank}_c(L^-.\ell, i)$, the interval of string cP_v in L^+ starts at $C[c] + j$, and this is exactly the lexicographic rank of (u, v) according to P_u, breaking ties according to P_v. We represent L^- and L^+, aligned and in compact form, using three arrays, `labels`, `out`, and `last`, as follows. For a vertex $v \in V$, let $[i..j]$ and $[i'..j']$ be its intervals in L^- and in L^+, respectively. Such intervals are possibly different, but both of them are the kth interval in their list for some k. We assign to every $v \in V$ *exactly one interval* $\mathbb{I}(v)$ of size $\max\{j - i + 1, j' - i' + 1\}$, and we set

$$\mathtt{labels}[\overleftarrow{v} + k] = \begin{cases} L^-[i + k].\ell & \text{for } k \in [0..j - i] \\ 0 & \text{otherwise,} \end{cases}$$

$$\mathtt{out}[\overleftarrow{v} + k] = \begin{cases} 1 & \text{for } k \in [0..j' - i'] \\ 0 & \text{otherwise.} \end{cases}$$

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

We also set a one in `last` at $\overrightarrow{v}$ for every $v \in V$. Note that the arrays `labels`, `out`, and `last` have each m components, where $m = \sum_{v \in V} \max\{|N^+(v)|, |N^-(v)|\} \leq 2|E| - (|V| - 2)$. The non-empty subarray of `labels` is an implicit representation of L^-, thus all arcs (u, v) with the same $\ell(u)$ appear in `labels` in lexicographic order of P_v. The non-empty subarray of `out` is the unary encoding of the *size* of the block of every P_v in L^+, which is the number of arcs $(v, w) \in E$. Analogously to L^- and L^+, we can easily move from the position i of an arc (u, v) in `labels` to its corresponding position in `out`, by computing

$$\mathsf{LF}(i, c) = \texttt{select}_1\big(\texttt{out}, C[c] + \texttt{rank}_c(\texttt{labels}, i)\big), \tag{9.5}$$

where $c = \texttt{labels}[i]$. We call this set of arrays the *Burrows–Wheeler index of the labeled, directed, acyclic graph G* (see Figure 9.9(c)).

DEFINITION 9.19 *Given a labeled, strongly distinguishable DAG G with n vertices and m arcs, the* Burrows–Wheeler index of *G, denoted by BWT_G, is a data structure that consists of*

- *the array of characters* `labels`*, encoded using a wavelet tree (see Section 3.3);*
- *bitvectors* `last` *and* `out`*, indexed to support* `rank` *and* `select` *queries in constant time as described in Section 3.2;*
- *the count array C.*

See the main text for the definition of such arrays. BWT_G takes $m \log \sigma (1 + o(1)) + 2m + o(m)$ bits of space.

Analogously to previous sections, we denote by $\overrightarrow{W}$ the range of BWT_G that contains the intervals of all vertices $v \in V$ whose distinguishing prefix P_v is prefixed by string $W \in [0..\sigma + 1]^+$, or the interval of the only vertex $v \in V$ whose P_v is a prefix of W. Note that, if G is a distinguishable tree, the arrays `labels` and `last` coincide with those of the Burrows–Wheeler index of a tree described in Section 9.5, up to the criterion used for breaking ties in the lexicographic order.

*9.6.1 Moving backward

Like its specialization for trees, BWT_G supports a number of navigational primitives. For example, assume that we know $\overrightarrow{v}$ for a specific vertex v, and that we want to determine its label $\ell(v)$. We know that the number of arcs (u, w) whose P_u is lexicographically smaller than P_v is $i = \texttt{rank}_1(\texttt{out}, \overleftarrow{v} - 1)$, thus $\ell(v)$ is the character $c \in [0..\sigma + 1]$ that satisfies $C[c] < i + 1 \leq C[c + 1]$, which can be determined by a binary search over C. Thus, computing $\ell(v)$ takes $O(\log \sigma)$ time.

Similarly, let function $\texttt{inNeighbor}(\overrightarrow{v}, c) = \overrightarrow{u}$ return the interval of the only in-neighbor u of v with $\ell(u) = c$. Assume that i is the position of c in $\texttt{labels}[\overrightarrow{u}]$, or equivalently that $i \in \overrightarrow{u}$ and $\texttt{labels}[i] = c$. Then, position $\mathsf{LF}(i, c)$ belongs to $\overrightarrow{u}$ in `out`, thus we can implement the function `inNeighbor` as follows:

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

$$\texttt{inNeighbor}(\overrightarrow{v}, c) = \Big[\texttt{select}_1\Big(\texttt{last}, \texttt{rank}_1\big(\texttt{last}, \mathsf{LF}(\overrightarrow{v}, c) - 1\big)\Big) + 1\ ..\ \texttt{select}_1\Big(\texttt{last}, \texttt{rank}_1\big(\texttt{last}, \mathsf{LF}(\overrightarrow{v}, c) - 1\big) + 1\Big)\Big]. \quad (9.6)$$

See Figure 9.9(c) for a concrete example. Note that this operation takes $O(\log \sigma)$ time.

Recall that the interval of a string $W \in [0..\sigma+1]^+$ in BWT_G contains the intervals of all vertices $v \in V$ such that W is a prefix of P_v, or the interval of the only vertex $v \in V$ such that P_v is a prefix of W. In the first case, the paths labeled by W in G are exactly the paths of length $|W|$ that start from a vertex v with $\overrightarrow{v} \subseteq \overrightarrow{W}$. In the second case, the paths labeled by W in G are a subset of all the paths of length $|W|$ that start from v. We can lift the function `inNeighbor` to map from $\overrightarrow{W}$ to $\overrightarrow{cW}$ for $c \in [1..\sigma+1]$, as follows:

$$\texttt{extendLeft}(c, \overrightarrow{W}, \overrightarrow{W}) = \Big[\texttt{select}_1\Big(\texttt{last}, \texttt{rank}_1\big(\texttt{last}, \mathsf{LF}(\overrightarrow{W} - 1, c)\big)\Big) + 1\ ..\ \texttt{select}_1\Big(\texttt{last}, \texttt{rank}_1\big(\texttt{last}, \mathsf{LF}(\overrightarrow{W}, c) - 1\big) + 1\Big)\Big]. \quad (9.7)$$

This procedure clearly takes $O(\log \sigma)$ time, and it supports a generalization of the backward search algorithm described in Section 9.2.2 that returns the interval of all vertices in V that are the starting point of a path labeled by a string W of length m: the search proceeds again from right to left along W, computing iteratively $\overrightarrow{W_{i..m}} = \texttt{extendLeft}(\overrightarrow{W_{i+1..m}}, W[i])$. More generally, the following lemma holds.

LEMMA 9.20 BWT_G *is a* forward BWT index *(Definition 9.13). The operations* `extendLeft` *and* `isLeftMaximal` *can be implemented in* $O(\log \sigma)$ *time, and* `enumerateLeft` *can be implemented in* $O(d \log(\sigma/d))$ *time, where* d *is the size of the output.*

*9.6.2 Moving forward

Let function $\texttt{outNeighbor}(\overrightarrow{v}, i) = \overrightarrow{w}$ be such that $w \in V$ is the out-neighbor of v whose P_w has lexicographic rank i among all the out-neighbors of v. Assume that $\ell(v) = c$. There are $j = \texttt{rank}_1(\texttt{out}, \overrightarrow{v} - 1)$ arcs (x, y) such that P_x is lexicographically smaller than P_v, and in particular there are $j - C[c]$ such arcs with P_x prefixed by c. It follows that position $k = \texttt{select}_c(\texttt{labels}, j - C[c] + i)$ in `labels` belongs to $\overrightarrow{w}$, thus

$$\begin{aligned} \texttt{outNeighbor}(\overrightarrow{v}, i) &= \Big[\texttt{select}_1\big(\texttt{last}, \texttt{rank}_1(\texttt{last}, k-1)\big) + 1\ ..\ \texttt{select}_1\big(\texttt{last}, \texttt{rank}_1(\texttt{last}, k-1) + 1\big)\Big], \\ k &= \texttt{select}_c\big(\texttt{labels}, \texttt{rank}_1(\texttt{out}, \overrightarrow{v} - 1) - C[c] + i\big). \\ c &= \ell(v) \end{aligned} \quad (9.8)$$

See Figure 9.9(c) for a concrete example. Note that this operation takes $O(\log \sigma)$ time, and that the formula to compute k is identical to Equation (9.4), which implements the

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

homologous function in the Burrows–Wheeler index of a tree. It is also easy to index numerical values assigned to vertices, such as identifiers or probabilities, by adapting the sampling approach described in Section 9.2.3: see Exercise 9.12.

*9.6.3 Construction

A labeled DAG might not be distinguishable, and it might not even be reverse-deterministic. It is possible to transform $G = (V, E, \Sigma)$ into a reverse-deterministic DAG $G' = (V', E', \Sigma)$ such that $\bar{\ell}(s') = \bar{\ell}(s)$, by applying the classical *powerset construction algorithm* for determinizing finite automata, described in Algorithm 9.4. Let A be an acyclic, nondeterministic finite automaton with n states that recognizes a finite language $L(A)$: in the worst case, the number of states of the minimal *deterministic* finite automaton that recognizes $L(A)$ is exponential in n, and this bound is tight. Therefore, Algorithm 9.4 can produce a set of vertices V' of size $O\left(2^{|V|}\right)$ in the worst case.

Algorithm 9.4: Powerset construction

Input: Directed labeled graph $G = (V, E, \Sigma)$, not necessarily acyclic.
Output: Directed, labeled, reverse-deterministic graph $G' = (V', E', \Sigma)$ such that $\bar{\ell}(s') = \bar{\ell}(s)$.

```
1  t' ← {t};
2  ℓ(t') = #;
3  V' ← {t'};
4  E' ← ∅;
5  toBeExpanded ← {t'};
6  expanded ← ∅;
7  while toBeExpanded ≠ ∅ do
8      Let A ∈ toBeExpanded;
9      for c ∈ Σ ∪ {$} do
10         A' ← {u ∈ V | (u, v) ∈ E, v ∈ A, ℓ(u) = c};
11         ℓ(A') ← c;
12         E' ← E' ∪ {(A', A)};
13         V' ← V' ∪ {A'};
14         if A' ∉ expanded then
15             toBeExpanded ← toBeExpanded ∪ {A'};
16         toBeExpanded ← toBeExpanded \ {A};
17         expanded ← expanded ∪ {A};
```

To make a labeled, reverse-deterministic DAG G' strongly distinguishable (or, more concretely, to transform graph G into graph H in Figure 9.9), we can adapt the prefix-doubling approach used in Lemma 8.8 for building suffix arrays. Specifically,

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

G' is said to be *k-sorted* if, for every vertex $v \in V'$, one of the following conditions holds:

- v is strongly distinguishable with $|P_v| \leq k$;
- all paths that start from v share the same prefix of length exactly k.

Starting from G', which is 1-sorted, it is possible to build a *conceptual* sequence of 2^i-sorted graphs $G^i = (V^i, E^i)$ for $i = 1, 2, \ldots, \lceil \log(p) \rceil$, where p is the length of a longest path in G'. The final graph $G^{\lceil \log(p) \rceil} = G^*$ is p-sorted, thus it is also strongly distinguishable.

Specifically, we project each vertex $v \in V'$ into *a set of vertices* in V^*, corresponding to the distinct prefixes of paths that starts from v in G', as follows. Assume that a vertex $v \in V^i$ is annotated with the following elements:

- $\texttt{path}(v)$, the string of length 2^i that prefixes all paths of G^i that start from v;
- $\texttt{source}(v)$, the original vertex in V' that was projected to v;
- $\texttt{targets}(v)$, the set of all vertices in V' that are reachable from $\texttt{source}(v)$ using a path of length 2^i labeled by $\texttt{path}(v)$.

Let L be the list that results from sorting set $\{\texttt{path}(v) : v \in V^i\}$ in lexicographic order, and let $R[1..|V^i|]$ be an array such that $R[v]$ is the position of the first occurrence of $\texttt{path}(v)$ in L. If $R[u]$ is unique in R, then vertex u is strongly distinguishable, and it is projected onto a single vertex in V^{i+1} with the same values of `path`, `source`, and `targets`. Otherwise, we add to V^{i+1} a vertex uvw for every pair of vertices $v \in V'$ and $w \in V^i$ such that $v \in \texttt{targets}(u)$ and $\texttt{source}(w) = v$, where $\texttt{path}(uvw) = \texttt{path}(u)[1..2^i - 1] \cdot \texttt{path}(w)$ and $\texttt{targets}(uvw) = \texttt{targets}(w)$. We then merge all vertices in V^{i+1} with identical `source` and `path` into a single vertex, whose `targets` field is the union of the `targets` fields of the merged vertices. Finally, we build the set of arcs E^* just for the final graph G^*: this set consists of all pairs $v, w \in V^*$ such that $(\texttt{source}(v), \texttt{source}(w)) \in E$ and $\texttt{path}(v)$ is a proper prefix of $\ell(\texttt{source}(v)) \cdot \texttt{path}(w)$. Note that this process expands the reverse-deterministic DAG G', and this DAG can in turn be exponential in the size of the input graph G. However, we still have that $|V^*| \in O(2^{|V|})$ in the worst case: see Exercise 9.15.

Note that G^* is strongly distinguishable only if graph G' is reverse-deterministic. Note also that a navigation on the original graph G can be simulated by a navigation on G^*. Building the vertices and arcs of G^* reduces to sorting, scanning, and joining lists of tuples of integers, thus it can be implemented efficiently in practice: see Exercise 9.16.

9.7 BWT indexes for de Bruijn graphs

A de Bruijn graph is a representation of the set of all distinct strings of a fixed length k, called *k-mers*, over a given alphabet $\Sigma = [1..\sigma]$.

DEFINITION 9.21 *Let k be an integer and let $\Sigma = [1..\sigma]$ be an alphabet. A* de Bruijn graph *with parameters Σ and k is a graph $\mathsf{DBG}_{\Sigma,k} = (V, E)$ with $|V| = \sigma^{k-1}$ vertices*

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

and $|E| = \sigma^k$ *arcs, in which every vertex is associated with a distinct string of length* $k-1$ *on* $[1..\sigma]$*, and every arc is associated with a distinct string of length* k *on* $[1..\sigma]$*. An arc* (v, w) *associated with* k*-mer* $c_1 \cdots c_k$ *connects the vertex* v *associated with string* $c_2 \cdots c_{k-1}c_k \in [1..\sigma]^{k-1}$ *to the vertex* w *associated with string* $c_1c_2 \cdots c_{k-1} \in [1..\sigma]^{k-1}$.

In this section we will systematically abuse notation, and identify a vertex of $\mathsf{DBG}_{\Sigma,k}$ with its associated $(k-1)$-mer, and an arc of $\mathsf{DBG}_{\Sigma,k}$ with its associated k-mer. Moreover, we will work with *subgraphs* of the full de Bruijn graph. Specifically, given a *cyclic string* T of length N on alphabet $[1..\sigma]$, which contains n distinct k-mers and m distinct $(k-1)$-mers, we are interested in the subgraph $\mathsf{DBG}_{T,k}$ of the de Bruijn graph $\mathsf{DBG}_{\Sigma,k}$ which contains m vertices and n arcs, such that every $(k-1)$-mer that appears in T has an associated vertex in $\mathsf{DBG}_{T,k}$, and every k-mer that appears in T has an associated arc in $\mathsf{DBG}_{T,k}$ (see the example in Figure 9.10(a)). Such graphs are the substrate of a number of methods to assemble fragments of DNA into chromosomes, a fundamental operation for reconstructing the genome of a new species from a set of reads (see Chapter 13). We will waive the subscripts from $\mathsf{DBG}_{T,k}$ whenever they are clear from the context, thus using DBG to refer to a subgraph of the full de Bruijn graph.

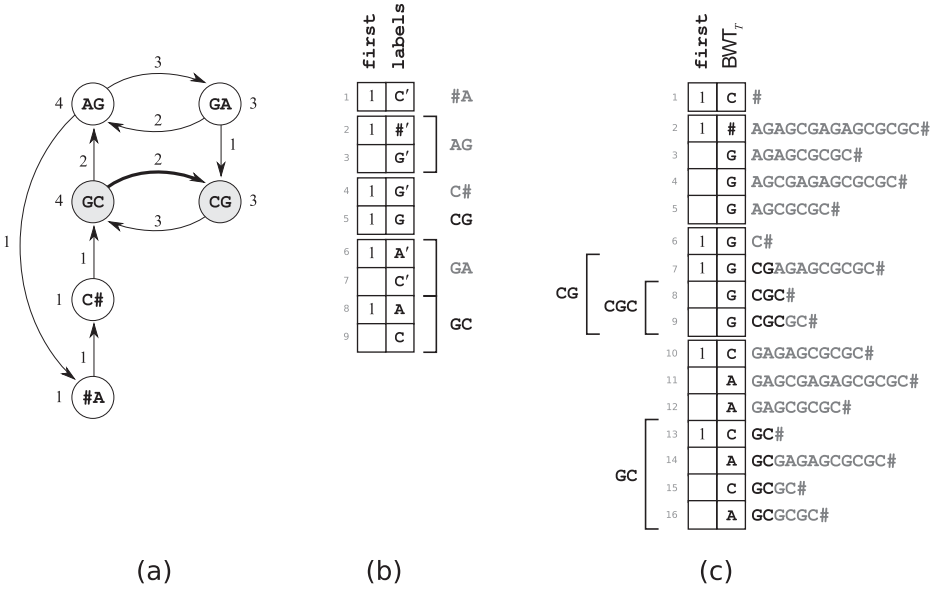

Figure 9.10 The de Bruijn graph $\mathsf{DBG}_{T,3}$ of the cyclic string $T = \texttt{AGAGCGAGAGCGCGC\#}$ (a), with its frequency-oblivious (b) and its frequency-aware (c) representations. Vertices and arcs in (a) are labeled by the frequency of the corresponding dimers and trimers in T, respectively. In panels (b) and (c), zeros are omitted from bitvectors, and spaces mark the starting position of the interval of every character in $[1..\sigma]$. All panels highlight the transition from vertex GC to vertex CG.

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

Let v be a vertex of $\mathsf{DBG}_{T,k}$ associated with $(k-1)$-mer $c_2 \cdots c_k$, where $c_i \in [1..\sigma]$ for $i \in [2..k]$. In this section we are interested in the following operations on $\mathsf{DBG}_{T,k}$.

`getArc`(v, c_1)	Return the unique identifier of the arc from v associated with k-mer $c_1 c_2 \cdots c_k$, where $c_1 \in [1..\sigma]$, or -1 if no such arc exists.
`followArc`(v, c_1)	If `getArc`$(v, c_1) \neq -1$, return the unique identifier of the vertex associated with $(k-1)$-mer $c_1 c_2 \cdots c_{k-1}$.
`outDegree`(v)	Return the number of outgoing arcs from vertex v.
`vertexFreq`(v)	Return the number of occurrences in T of the $(k-1)$-mer associated with vertex v.
`arcFreq`(e)	Return the number of occurrences in T of the k-mer associated with arc e.

We assume that vertex and arc identifiers occupy a constant number of machine words, or equivalently $O(\log N)$ bits of space. A naive representation of $\mathsf{DBG}_{T,k}$ would clearly take $O(n \log n)$ bits: in this section we describe two alternative representations that take significantly less space, implement `getArc` and `followArc` in $O(\log \sigma)$ time, and can be built efficiently from the bidirectional Burrows–Wheeler index of T by adapting Algorithm 9.3.

Table 9.1 provides an overview of the two representations. Note that the frequency-aware representation takes more space than the frequency-oblivious representation whenever $n \ll N$, or equivalently whenever string T is highly repetitive. However, we will see that the frequency-aware representation is more flexible, since building it for different values of k requires just creating a different bitvector per value (see Section 9.7.3). In what follows we will also see that the two representations become essentially identical when every k-mer occurs at most once in T: see Exercise 9.21. It will also become apparent how to implement the function `enumerateLeft`, making both representations essentially equivalent to a forward Burrows–Wheeler index.

Without loss of generality, in what follows we will assume that the last character of T is the special character $0 = \# \notin [1..\sigma]$, and we will reset Σ to $\Sigma \cup \{\#\}$, σ to $\sigma + 1$, and N to $N + 1$. For brevity, we will also say that arc $e = c_1 c_2 \cdots c_{k-1}$ from vertex $v = c_2 \cdots c_{k-1}$ *is labeled by character* c_1. We will also focus on the case in which T is a single string: adapting the representations to the case in which T is the concatenation of a set of strings is left to Exercise 9.20.

9.7.1 Frequency-oblivious representation

In this representation the identifier of a vertex v is the lexicographic rank of the $(k-1)$-mer associated with v, among all distinct $(k-1)$-mers that occur in T. This is equivalent to what has been described for labeled trees in Section 9.5. Similarly, the identifier of an arc e is the lexicographic rank of the k-mer associated with e, among all distinct k-mers that occur in T. Note that the lexicographic order induces a total order on the vertices and on the arcs of $\mathsf{DBG}_{T,k}$: let $v_1, v_2, \ldots, v_m$ be the vertices of $\mathsf{DBG}_{T,k}$ in lexicographic order.

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

Table 9.1 Overview of the representations of $\mathsf{DBG}_{T,k}$ described in Section 9.7. N is the length of T, n is the number of distinct k-mers that occur in T, σ is the cardinality of the alphabet, and d is the size of the output of operation `outDegree`.

	Frequency-oblivious	**Frequency-aware**
Space (bits)	$n(2+\log\sigma(1+o(1))$	$N(1+\log\sigma(1+o(1))$
Construction time	$O(N\log\sigma)$	$O(N\log\sigma)$
`getArc`	$O(\log\sigma)$	$O(\log\sigma)$
`followArc`	$O(\log\sigma)$	$O(\log\sigma)$
`outDegree`	$O(1)$	$O(d\log(\sigma/d))$
`vertexFreq`		$O(1)$
`arcFreq`		$O(1)$

To represent DBG we apply the same strategy as described in Section 9.5 for labeled trees: namely, we build the sorted list $v_1, v_2, \ldots, v_m$, we allocate an array `labels`$[1..n]$ of characters from alphabet $[1..\sigma]$, and we store the labels of the n_i arcs that start from vertex v_i, in arbitrary order, inside interval $\overrightarrow{v_i}$ of `labels`, where $\overrightarrow{v_i} = \sum_{p=1}^{i-1} n_p + 1$ and $\overleftarrow{v_i} = \sum_{p=1}^{i} n_p$. Thus, we assign to each vertex $v \in V$ an interval $\overrightarrow{v}$ in array `labels`. We represent `labels` as a wavelet tree, so that it supports `rank` and `select` queries, and we use again a bitvector `first`$[1..n]$ to mark the first position of every interval of a vertex: specifically, `first`$[i] = 1$ if and only if $i = 1$ or the arc with identifier i has a different source vertex from the arc with identifier $i-1$. We index `first` to support `rank` and `select` queries in constant time. Finally, we use an array $C_k[1..\sigma]$ that stores at position $C_k[c]$ the number of distinct k-mers prefixed by characters smaller than c, as well as an array $C_{k-1}[1..\sigma]$ that stores at position $C_{k-1}[c]$ the number of distinct $(k-1)$-mers prefixed by characters smaller than c.

Let i be the lexicographic rank of the $(k-1)$-mer $c_2 \cdots c_k$ associated with a vertex v of DBG. To compute $\overrightarrow{v}$, it suffices to perform

$$\overleftarrow{v} = \texttt{select}_1(\texttt{first}, i), \tag{9.9}$$

$$\overrightarrow{v} = \texttt{select}_1(\texttt{first}, i+1) - 1; \tag{9.10}$$

and to get `outDegree`$(\overrightarrow{v})$ it suffices to compute $|\ \overrightarrow{v}\ |$. To check whether there is an arc $e = (v, w)$ labeled by character c_1, we just need to check whether `labels`$[\overrightarrow{v}]$ contains character c_1, that is, whether $p = \texttt{rank}_{c_1}(\texttt{labels}, \overrightarrow{v})$ is greater than $\texttt{rank}_{c_1}(\texttt{labels}, \overleftarrow{v} - 1)$. If this is the case, we return

$$\texttt{getArc}(\overrightarrow{v}, c_1) = C_k[c_1] + \texttt{rank}_{c_1}(\texttt{labels}, \overleftarrow{v} - 1) + 1, \tag{9.11}$$

which is indeed the lexicographic rank of the k-mer $c_1 c_2 \cdots c_k$ associated with e among all k-mers that occur in T.

To implement `followArc`$(\overrightarrow{v}, c_1)$, we need to return the lexicographic rank of the $(k-1)$-mer $c_1 c_2 \cdots c_{k-1}$: this rank is $C_{k-1}[c_1] + q$, where q is the number of $(k-2)$-mers that are lexicographically smaller than or equal to $c_2 \cdots c_{k-1}$, *and that are the origin of an arc labeled by* c_1. Note that a string W of length $k-2$ corresponds to

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

the contiguous interval $\overrightarrow{W}$ in `labels` that contains all vertices whose $(k-1)$-mer is prefixed by W. Consider thus $\overrightarrow{W}$, and let $i_1, i_2, \ldots, i_r$ be all the occurrences of character c_1 in `labels`$[\overrightarrow{W}]$, or equivalently all the arcs that start from a vertex prefixed by W and that end in a vertex prefixed by c_1W. Assume that we preventively marked `labels`$[i_1]$ by setting it to an artificial character c_1' rather than to c_1, and that we performed this marking for every $(k-2)$-mer W and for every character in $[1..\sigma]$. Note that there are exactly m marked positions in total, where m is the number of distinct $(k-1)$-mers in T. Then, we return

$$\texttt{followArc}(\overrightarrow{v}, c_1) = C_{k-1}[c_1] + q,$$
$$q = \texttt{rank}_{c_1'}(\texttt{labels}, 1, \overrightarrow{v}). \tag{9.12}$$

Supporting `followArc` requires `labels` to store characters from alphabet $\Sigma \cup \Sigma'$, where Σ' is of size σ and contains a character c' for every character $c \in \Sigma$. Adapting the implementation of `getArc` to take characters in Σ' into account is left to Exercise 9.18.

The data structures described in this section represent just the *topology* of DBG, therefore all of the information about the frequency of $(k-1)$-mers and of k-mers is lost. A possible way to support `vertexFreq` and `arcFreq` consists in *indexing the original string T rather than* DBG, as described below.

9.7.2 Frequency-aware representation

In this representation the identifier of a vertex is the interval of the corresponding $(k-1)$-mer in BWT_T, and the identifier of an arc is the interval of the corresponding k-mer in BWT_T. Note that, for any k, the set of intervals of all k-mers is a *partition* of the range $[1..N]$, that is, each position in $[1..N]$ belongs to the interval of exactly one k-mer. Note also that the interval of a k-mer $c_1c_2\cdots c_k$ either coincides with the interval of $(k-1)$-mer $c_1c_2\cdots c_{k-1}$ (if $c_1c_2\cdots c_{k-1}$ is always followed by character c_k in T), or is included in the interval of $c_1c_2\cdots c_{k-1}$.

Suppose that we are given the BWT of string T represented as a wavelet tree. We build a bitvector `first`$[1..N]$ in which we mark with a one the first position of the interval of every $(k-1)$-mer W that occurs in T, and we keep the other positions to zero. That is, `first`$[\overrightarrow{W}] = 1$, `first`$[i] = 0$ for all $i \in [\overrightarrow{W}+1..\overrightarrow{W}]$, and either $\overrightarrow{W} = N$ (if W is the last $(k-1)$-mer in lexicographic order) or `first`$[\overrightarrow{W}+1] = 1$. We then index `first` to support `rank` and `select` queries in constant time. Our representation of DBG consists of BWT_T, `first`, and the usual array $C[1..\sigma]$ which stores in $C[c]$ the number of occurrences of characters smaller than c in T.

Associating vertices and arcs with intervals of BWT_T makes it easy to implement the following operations on $\mathsf{DBG}_{T,k}$:

$$\texttt{vertexFreq}(\overrightarrow{v}) = |\,\overrightarrow{v}\,|, \tag{9.13}$$
$$\texttt{arcFreq}(\overrightarrow{e}) = |\,\overrightarrow{e}\,|, \tag{9.14}$$
$$\texttt{outDegree}(\overrightarrow{v}) = |\,\texttt{rangeList}(\mathsf{BWT}_T, \overrightarrow{v}, \overrightarrow{v}, 1, \sigma)\,|, \tag{9.15}$$

where `rangeList` is the $O(d\log(\sigma/d))$-time function provided by the wavelet tree of BWT_T (see Section 3.3.2). Implementing `getArc` is equally straightforward, since it

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

amounts to performing a $O(\log \sigma)$-time backward step on BWT_T with character c_1:

$$\begin{aligned} \texttt{getArc}(\overleftarrow{v}, c_1) = \overrightarrow{e}, \\ \overleftarrow{e} = C[c_1] + \texttt{rank}_{c_1}(\mathsf{BWT}_T, \overleftarrow{v} - 1) + 1, \\ \overrightarrow{e} = C[c_1] + \texttt{rank}_{c_1}(\mathsf{BWT}_T, \overrightarrow{v}). \end{aligned} \tag{9.16}$$

Assume that vertex v corresponds to the $(k-1)$-mer $c_2 \cdots c_k$. To implement $\texttt{followArc}(\overleftarrow{v}, c_1)$ we first call $\texttt{getArc}(\overleftarrow{v}, c_1)$ to obtain the interval of the arc $e = (v, w)$ labeled by c_1 that starts from v. Note that e corresponds to k-mer $c_1 \cdots c_k$, and we want to compute the interval of vertex w associated with the $(k-1)$-mer $c_1 \cdots c_{k-1}$. Clearly $\overrightarrow{e} \subseteq \overrightarrow{w}$, therefore we know that $\texttt{first}[i] = 0$ for all $i \in [\overleftarrow{e} + 1..\overrightarrow{e}]$. If $\texttt{first}[\overleftarrow{e}] = 1$ and either $\overrightarrow{e} = N + 1$ or $\texttt{first}[\overrightarrow{e} + 1] = 1$, then $\overrightarrow{e} = \overrightarrow{w}$: this happens whenever string $c_1 \cdots c_{k-1}$ is always followed by character c_k in T. Otherwise, $\overrightarrow{e} \subset \overrightarrow{w}$. If $\texttt{first}[\overleftarrow{e}] = 1$ then clearly $\overleftarrow{w} = \overleftarrow{e}$. Otherwise $\overleftarrow{w} < \overleftarrow{e}$, the interval $\texttt{first}[\overleftarrow{w} + 1..\overleftarrow{e} - 1]$ contains only zeros, and $\overleftarrow{w}$ is the position of the last one in $\texttt{first}$ that precedes position $\overleftarrow{e}$:

$$\overleftarrow{w} = \texttt{select}_1\big(\texttt{first}, \texttt{rank}_1(\texttt{first}, \overleftarrow{e})\big). \tag{9.17}$$

Similarly, if $\overrightarrow{e} = N$ then $\overrightarrow{w} = \overrightarrow{e}$, otherwise $\overrightarrow{w} + 1$ is the first one which follows position $\overrightarrow{e}$ in $\texttt{first}$:

$$\overrightarrow{w} = \texttt{select}_1\big(\texttt{first}, \texttt{rank}_1(\texttt{first}, \overrightarrow{e}) + 1\big) - 1. \tag{9.18}$$

9.7.3 Space-efficient construction

Both the frequency-aware and the frequency-oblivious representations of $\mathsf{DBG}_{T\#,k}$ for some string $T \in [1..\sigma]^{N-1}$ can be built from $\mathsf{BWT}_{T\#}$, augmented with the bitvector $\texttt{first}_k[1..N]$ which marks the starting position of every interval in $\mathsf{BWT}_{T\#}$ that corresponds to a k-mer, for some value of k. It turns out that $\texttt{first}_k$ itself can be derived from the bidirectional BWT index of T.

LEMMA 9.22 *Given an integer k, and the bidirectional BWT index of a string T of length $N-1$ represented as in Theorem 9.14, we can compute array $\texttt{first}_k$ in $O(N \log \sigma)$ time and in $O(\sigma \log^2 N)$ bits of working space.*

Proof To build $\texttt{first}$ for a given value of k, we start by setting $\texttt{first}[i] = 1$ for all $i \in [1..N]$. Then, we use Algorithm 9.3 to enumerate all intervals that correspond to an internal node v of the suffix tree of T, as well as the length of label $\ell(v)$, in overall $O(N \log \sigma)$ time. For every internal node v we check whether $|\ell(v)| \geq k$: if this is the case, we use operations $\texttt{enumerateRight}$ and $\texttt{extendRight}$ provided by the bidirectional BWT index of T to compute the intervals of all the children $w_1, w_2, \ldots, w_h$ of v in the suffix tree of T, in lexicographic order. Then, we flip bit $\texttt{first}[\overleftarrow{w_i}]$ for all $i \in [2..h]$. Proving that this algorithm builds $\texttt{first}_k$ is left to Exercise 9.19. □

If the bidirectional BWT index of T is provided in the input, we already have $\mathsf{BWT}_{T\#}$ represented as a wavelet tree, as well as the vector $C[0..\sigma]$. Thus, to build the

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

frequency-aware representation of $\mathsf{DBG}_{T\#,k}$, we just need to index $\texttt{first}_{k-1}$ to answer `rank` and `select` queries in constant time.

THEOREM 9.23 *Given the bidirectional BWT index of a string T of length $N-1$ represented as in Theorem 9.14, we can build the frequency-aware representation of the de Bruijn graph $\mathsf{DBG}_{T\#,k}$ in $O(N \log \sigma)$ time and in $O(\sigma \log^2 N)$ bits of working space.*

Building the frequency-oblivious representation of $\mathsf{DBG}_{T\#,k}$ requires just a little more effort.

THEOREM 9.24 *Given the bidirectional BWT index of a string T of length $N-1$ represented as in Theorem 9.14, we can build the frequency-oblivious representation of the de Bruijn graph $\mathsf{DBG}_{T\#,k}$ in $O(N \log \sigma)$ time and in $O(N + \sigma \log^2 N)$ bits of working space.*

Proof We build $\texttt{first}_{k-1}$ as described in Lemma 9.22. To avoid ambiguity, we rename $\texttt{first}_{k-1}$ to B_{k-1}. Then, we initialize the bitvector `first` of the frequency-oblivious representation of $\mathsf{DBG}_{T\#,k}$ to all zeros, except for $\texttt{first}[1] = 1$. Let p_1, p_2, and p_3 be pointers to a position in B_{k-1}, in $\mathsf{BWT}_{T\#}$, and in `labels`, respectively, and assume that they are all initialized to one. The first one in B_{k-1} is clearly at position one: we thus move p_1 from position two, until $B_{k-1}[p_1] = 1$ (or until $p_1 > N$). Let W be the first $(k-1)$-mer in lexicographic order. The interval of W in $\mathsf{BWT}_{T\#}$ is $[1..p_1 - 1]$. We can thus initialize a bitvector $\texttt{found}[0..\sigma]$ to all zeros, move pointer p_2 between one and $p_1 - 1$ over $\mathsf{BWT}_{T\#}$, and check whether $\texttt{found}[\mathsf{BWT}[p_2]] = 0$: if so, we set $\texttt{labels}[p_3] = \mathsf{BWT}[p_2]$, we increment p_3 by one, and we set $\texttt{found}[\mathsf{BWT}[p_2]]$ to one. At the end of this process, all distinct characters that occur to the left of W in $T\#$ have been collected in the contiguous interval $[1..p_3 - 1]$ of `labels`, so we can set $\texttt{first}[p_3] = 1$. To reinitialize `found` to all zeros, we scan again the interval $[1..p_3 - 1]$ by moving a pointer q, and we iteratively set $\texttt{found}[\texttt{labels}[q]] = 0$. We repeat this process with the following blocks marked in B_{k-1}.

The marking of the first occurrence of every distinct character that appears in the interval of a $(k-2)$-mer can be embedded in the algorithm we just described. Specifically, we first build the bitvector $\texttt{first}_{k-2}$ using again the algorithm described in Lemma 9.22, and we rename it B_{k-2}. Then, we keep the first and last position of the current interval of a $(k-2)$-mer in the variables x and y, respectively. Whenever we are at the end of the interval of a $(k-2)$-mer, or equivalently whenever $p_1 > N$ or $B_{k-2}[p_1] = 1$, we set $y = p_1 - 1$ and we scan again $\texttt{labels}[x..y]$, using the array `found` as before to detect the first occurrence of every character. At the end of this scan, we reset $x = y + 1$ and we repeat the process.

Building arrays C_k and C_{k-1} is easy, and it is left to the reader. □

9.8 Literature

The Burrows–Wheeler transform was first introduced by Burrows & Wheeler (1994) as a tool for text compression. It was immediately observed that the construction of the

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

transform is related to suffix arrays, but it took some time for the tight connection that enables backward search in compressed space to be discovered in Ferragina & Manzini (2000) and Ferragina & Manzini (2005). In the literature the BWT index is often called the *FM-index*. Simultaneously, other ways of compressing suffix arrays were developed (Grossi & Vitter 2000, 2006; Sadakane 2000) building on the inverse of the function LF, called ψ therein. Our description of succinct suffix arrays follows Navarro & Mäkinen (2007).

The space-efficient construction of the Burrows–Wheeler transform given in Section 9.3 is a variant of an algorithm that appears in Hon *et al.* (2003). The original method allows one to build the Burrows–Wheeler transform in $O(n \log\log \sigma)$ time using $O(n \log \sigma)$ bits of space. Our version takes more time because it uses wavelet trees, while the original method applies more advanced structures to implement `rank`. The merging step of the algorithm can be further improved by exploiting the simulation of a bidirectional BWT index with just one BWT (Belazzougui 2014): this gives an optimal $O(n)$ time construction using $O(n \log \sigma)$ bits. The original result uses randomization, but this has been improved to deterministic in the full version of the article.

The bidirectional BWT index described in Section 9.4 is originally from Schnattinger *et al.* (2010). A similar structure was proposed also in Li *et al.* (2009). The variant that uses just one BWT is similar to the one in Beller *et al.* (2012) and Beller *et al.* (2013), and it is described in Belazzougui (2014). The latter version differs from the original bidirectional BWT index in that it uses a stack that occupies $o(n)$ bits, rather than a queue and an auxiliary bitvector that take $O(n)$ bits of space as proposed in the original work. The use of the so-called *stack trick* to enumerate intervals that correspond to suffix tree nodes with the bidirectional BWT index in small space is from Belazzougui *et al.* (2013): in this chapter we extended its use to the variant in Belazzougui (2014) which uses only one BWT index. The stack trick itself has been used as early as the quicksort algorithm (Hoare 1962). The idea of detecting right-maximality using a bitvector to encode runs of equal character in the Burrows–Wheeler transform is due to Kulekci *et al.* (2012).

Extensions of the Burrows–Wheeler index to trees and graphs were proposed by Ferragina *et al.* (2009) and by Sirén *et al.* (2011) and Sirén *et al.* (2014). Exercise 9.10 is solved in Ferragina *et al.* (2009), and Exercises 9.12, 9.13, 9.17, and partially 9.16, are solved in Sirén *et al.* (2011) and Sirén *et al.* (2014). The Rabin–Scott powerset construction related to the latter is from Rabin & Scott (1959). The frequency-oblivious and frequency-aware representations of the de Bruijn graph are from Bowe *et al.* (2012) and Välimäki & Rivals (2013), respectively. The space-efficient algorithm for constructing the frequency-oblivious representation is a new contribution of this book. In Välimäki & Rivals (2013), a different space-efficient construction for our frequency-aware representation is shown. There are at least two other small-space representations of de Bruijn graphs based on the Burrows–Wheeler transform in Rødland (2013) and Chikhi *et al.* (2014). Yet other small-space representations are based on hashing (Chikhi & Rizk 2012; Pell *et al.* 2012; Salikhov *et al.* 2013), and more precisely on Bloom filters: these are not as space-efficient as the ones based on the Burrows–Wheeler transform.

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

The size of the genome of *Picea glauca* was taken from the databases of the National Center for Biotechnology Information, US National Library of Medicine, and the estimate for the genome of *Paris japonica* was derived from Pellicer *et al.* (2010).

Exercises

9.1 Consider the original definition of the Burrows–Wheeler transform with the cyclic shifts, without a unique endmarker # added to the end of the string. Assume you have a fast algorithm to compute our default transform that assumes the endmarker is added. Show that you can feed that algorithm an input such that you can extract the original cyclic shift transform efficiently from the output.

9.2 Describe an algorithm that builds the succinct suffix array of a string T from the BWT index of T, within the bounds of Theorem 9.6.

9.3 Consider the strategy for answering a set of `occ` locate queries in batch described in Section 9.2.4. Suppose that, once all positions in $\mathsf{SA}_{T\#}$ have been marked, a bitvector `marked`$[1..n+1]$ is indexed to answer rank queries. Describe an alternative way to implement the batch extraction within the same asymptotic time bound as the one described in Section 9.2.4, but *without resorting to radix sort*. Which strategy is more space-efficient? Which one is faster? The answer may vary depending on whether `occ` is small or large.

9.4 Recall that at the very end of the construction algorithm described in Section 9.3 we need to convert the BWT of the padded string X into the BWT of string $T\#$. Prove that $\mathsf{BWT}_X = T[n] \cdot \#^{k-1} \cdot W$ and $\mathsf{BWT}_{T\#} = T[n] \cdot W$, where $n = |T|$, $k = |X| - n$, and string W is a permutation of $T[1..n-1]$.

9.5 Assume that you have a machine with P processors that share the same memory. Adapt Algorithm 9.3 to make use of all P processors, and estimate the resulting speedup with respect to Algorithm 9.3.

9.6 Recall that Property 1 is key for implementing Algorithm 9.3 with just one BWT. Prove this property.

9.7 In many applications, the operation `enumerateLeft`(i,j), where $[i..j] = \mathbb{I}(W,T)$, is immediately followed by operations `extendLeft` $\big(c, \mathbb{I}(W,T), \mathbb{I}(\underline{W},\underline{T})\big)$ applied to every distinct character c returned by `enumerateLeft`(i,j).

i. Show how to use the operation `rangeListExtended`(T,i,j,l,r) defined in Exercise 3.12 to efficiently support a new operation called `enumerateLeftExtended`$(\mathbb{I}(W,T), \mathbb{I}(\underline{W},\underline{T}))$ that returns the distinct characters that appear in substring $\mathsf{BWT}_{T\#}[i..j]$, and for each such character c returns the pair of intervals computed by `extendLeft` $\big(c, \mathbb{I}(W,T), \mathbb{I}(\underline{W},\underline{T})\big)$. Implement also the symmetric operation `enumerateRightExtended`.

ii. Reimplement Algorithm 9.3 to use the operations `enumerateLeftExtended` and `enumerateRightExtended` instead of `enumerateLeft`, `enumerateRight`, `extendLeft`, and `extendRight`.

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

9.8 Recall that Property 2 is key for navigating the Burrows–Wheeler index of a labeled tree top-down (Section 9.5.1). Prove this property.

9.9 Consider the Burrows–Wheeler index of a tree $\mathcal{T}$ described in Section 9.5.

i. Given a character $c \in [1..\sigma]$, let $\overset{\bullet}{c}$ be the starting position of its interval $\overset{\bullet}{c}$ in $\mathsf{BWT}_{\mathcal{T}}$, and let $C'[1..\sigma]$ be an array such that $C'[c] = \overset{\bullet}{c}$. Describe an $O(n)$-time algorithm to compute C' from C and `last`.

ii. Recall that $\mathbb{I}(v) = [\overset{\bullet}{v}..\overset{\bullet}{v}]$ is the interval in $\mathsf{BWT}_{\mathcal{T}}$ that contains all the children of node v, and that $\mathbb{I}'(v)$ is the position of the arc $(v, \texttt{parent}(v))$ in $\mathsf{BWT}_{\mathcal{T}}$. Let $D[1..n+m-1]$ be an array such that $D[i] = \overset{\bullet}{v}$, where $\mathbb{I}'(v) = i$, or -1 if v is a leaf. Describe a $O(n+m)$-time algorithm to compute D from C', `last`, and `labels`. *Hint.* Use the same high-level strategy as in Lemmas 8.7 and 9.9.

iii. Give the pseudocode of an $O(n)$-time algorithm that reconstructs the original tree $\mathcal{T}$ from $\mathsf{BWT}_{\mathcal{T}}$ and D, in depth-first order.

9.10 In order to build the Burrows–Wheeler index of a tree $\mathcal{T}$ described in Section 9.5, we can adapt the prefix-doubling approach used in Lemma 8.8 for building suffix arrays. In particular, assume that we want to assign to every node v the order $R(v)$ of its path to the root, among all such paths in $\mathcal{T}$. We define the ith *contraction* of tree $\mathcal{T}$ as the graph $\widetilde{T}^i = (V, E^i)$ such that $(u, v) \in E^i$ iff $v_1, v_2, \ldots, v_{2^{i+1}}$ is a path in $\mathcal{T}$ with $v_0 = u$ and $v_{2^{i+1}} = v$. Clearly $\widetilde{T}^0 = \mathcal{T}$, and E^{i+1} is the set of all pairs (u, v) such that $P = u, w, v$ is a path in $\widetilde{T}^i$. Show how to iteratively refine the value $R(v)$ for every node, starting from the initial approximation $R^0(v) = \overset{\bullet}{\ell(v)} = C'[\ell(v)]$ computed as in Exercise 9.9. Describe the time and space complexity of your algorithm.

9.11 Assume that we want to generalize the recursive, $O(\sigma + n)$-time algorithm for building the suffix array $\mathsf{SA}_{S\#}$ of a string S to a similar algorithm for building the Burrows–Wheeler index of a labeled tree $\mathcal{T}$ described in Section 9.5. Do the results in Definition 8.9, Lemma 8.10, and Lemma 8.11 still hold? Does the overall generalization still achieve linear time for any $\mathcal{T}$? If not, for which topologies or labelings of $\mathcal{T}$ does the generalization achieve linear time?

9.12 Assume that a numerical value $\texttt{id}(v)$ must be associated with every vertex v of a labeled, strongly distinguishable DAG G: for example, $\texttt{id}(v)$ could be the probability of reaching vertex v from s. Assume that, whenever there is exactly one path P connecting vertex u to vertex v, we have that $\texttt{id}(v) = f(\texttt{id}(u), |P|)$, where f is a known function. Describe a method to augment the Burrows–Wheeler index of G described in Section 9.6 with the values of `id`, and its space and time complexity. *Hint.* Adapt the strategy used for strings in Section 9.2.3.

9.13 Consider the Burrows–Wheeler index of a labeled, strongly distinguishable DAG described in Section 9.6, and assume that `labels` is represented with $\sigma + 1$ bitvectors $B_c[1..m]$ for all $c \in [1..\sigma + 1]$, where $B_c[i] = 1$ if and only if $\texttt{labels}[i] = c$. Describe a more space-efficient encoding of `labels`. Does it speed up the operations described in Section 9.6?

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press

9.14 Given any directed labeled graph $G = (V, E, \Sigma)$, Algorithm 9.4 builds a reverse-deterministic graph $G' = (V', E', \Sigma)$ in which a vertex $v \in V'$ corresponds to *a subset of* V. Adapt the algorithm so that a vertex $v \in G'$ corresponds to *a subset of* E. Show an example in which the number of vertices in the reverse-deterministic graph produced by this version of the algorithm is smaller than the number of vertices in the reverse-deterministic graph produced by Algorithm 9.4.

9.15 Recall that in Section 9.6.3 a labeled DAG G is transformed into a reverse-deterministic DAG G', and then G' is itself transformed into a strongly distinguishable DAG G^*. Show that $|G^*| \in O(2^{|G|})$ in the worst case.

9.16 Consider the algorithm to enforce distinguishability described in Section 9.6.3.

i. Show that this algorithm does not also enforce reverse-determinism.
ii. Assume that the input to this algorithm contains two arcs (u, w) and (v, w) such that $\ell(u) = \ell(v)$: give an upper bound on the number of vertices that result from the projection of u and v to the output.
iii. Modify the algorithm to build a distinguishable, rather than a strongly distinguishable, DAG.
iv. Modify the algorithm to decide whether the input graph is distinguishable or not.
v. Implement the algorithm using just sorts, scans, and joins of lists of tuples of integers, *without storing or manipulating strings* `path` *explicitly*. Give the pseudocode of your implementation.
vi. Give the time and space complexity of the algorithm as a function of the size of its input and its output.

9.17 Describe a labeled, strongly distinguishable DAG that recognizes an infinite language. Describe an infinite language that cannot be recognized by any labeled, strongly distinguishable DAG.

9.18 Consider the frequency-oblivious representation of a de Bruijn graph described in Section 9.7.1. Show how to implement the function `getArc` when `labels` contains characters in $\Sigma \cup \Sigma'$.

9.19 Recall that marking the starting position of every interval in $\mathsf{BWT}_{T\#}$ that corresponds to a k-mer is a key step for building both the frequency-aware and the frequency-oblivious representation of a de Bruijn graph. Prove that Lemma 9.22 performs this marking correctly.

9.20 Some applications in high-throughput sequencing require the de Bruijn graph of *a set of strings* $\mathcal{R} = \{R^1, R^2, \ldots, R^r\}$. Describe how to adapt the data structures in Sections 9.7.1 and 9.7.2 to represent the de Bruijn graph of string $R = R^1 \# R^2 \# \cdots \# R^r \#$, where $\# \notin [1..\sigma]$. Give an upper bound on the space taken by the frequency-oblivious and by the frequency-aware representation.

9.21 Describe the frequency-oblivious and the frequency-aware representations of the de Bruijn graph $\mathsf{DBG}_{T,k}$ when every string of length k on a reference alphabet Σ occurs at most once in T.

https://doi.org/10.1017/CBO9781139940023.011 Published online by Cambridge University Press