11 Genome analysis and comparison

Aligning whole genomes using optimal dynamic programming algorithms is a daunting task, being practically feasible only for very similar species, and conceptually incapable of capturing large-scale phenomena that alter the contiguity of homologous regions, like chromosome-level rearrangements, gene shuffling and duplication, translocations, and inversions of large areas.

We could circumvent these limits by first using good local alignments as anchors and then finding a set of large-scale edit operations that align such anchors as well as possible: Exercise 11.8 elaborates on how to find these optimal one-to-one correspondences.

Alternatively, we could try to detect a set of genes that are shared by two species, and we could compute the minimum set of rearrangements that transform one sequence of genes into the other. Despite having the advantage of providing a constructive explanation of the distance between two genomes, this approach is feasible only for closely related species, it discards information contained in non-coding regions, it assumes that a large-enough set of common genes can be reliably identified in each genome and mapped across genomes, and it is ineffective in cases in which gene order is preserved, like in mammalian mitochondrial DNA.

Yet another alternative could be aligning just a few conserved genes, building a phylogenetic tree for each such gene, and merging the trees into a common consensus: this is practically difficult in some viral and bacterial families with high rates of mutation or lateral gene transfer, and it is conceptually undesirable since different genes can tell different phylogenetic stories. Alignment costs, moreover, have an intrinsic ambiguity.

In *alignment-free genome comparison* the goal is to derive efficiently computable distance or similarity measures for whole genomes that capture the relatedness without resorting to alignments. Such distance measures are typically derived between sets of *local features* extracted from the genomes. This allows one to compare genomes on the basis of their local compositional biases, rather than on the basis of their large-scale sequential structure: such compositional biases are widespread in nature, and have been shown to correlate with accepted species classifications. These distance measures generalize to comparing sets of unassembled reads, or mixed datasets containing both assembled and unassembled reads.

The chapter starts with the analysis of simple (shared) local features such as maximal repeats, maximal exact matches (MEMs), and maximal unique matches (MUMs), which are common building blocks of practical whole-genome alignment tools: see

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

Insight 11.1 for a simple example. The second half extends the techniques to alignment-free genome comparison methods.

Insight 11.1 MUMs as anchors for multiple alignment

Let us consider a high-throughput heuristic method for multiple alignment exploiting the maximal unique matches (MUMs) studied in Section 8.4.2. We assume that the genomes to be aligned are quite similar; that is, we assume that there is no need for global genome rearrangement operations to explain the differences. Let $A^1, A^2, \ldots, A^d$ be the sequences of length n_i for $1 \leq i \leq d$. The approach uses a divide-and-conquer strategy. Find the maximum-length MUM, say α, shared by all sequences. This MUM has exactly one location in each sequence, say j_i for $1 \leq i \leq d$, so it can be used for splitting the set into two independent parts: $A^1_{1\ldots j_1}, A^2_{1\ldots j_2}, \ldots, A^d_{1\ldots j_d}$ and $A^1_{j_1+|\alpha|\ldots n_1}, A^2_{j_2+|\alpha|\ldots n_2}, \ldots, A^d_{j_d+|\alpha|\ldots n_d}$. Now apply the same recursively for each part until sufficiently short sequences are obtained in each part; then apply the optimal multiple alignment algorithm for each part. The MUMs calculated this way work as anchors for the multiple alignment.

Many of our algorithms are modifications of Algorithm 9.3 on page 174 for simulating traversal over all suffix tree nodes using the bidirectional BWT index.

11.1 Space-efficient genome analysis

Recall the suffix tree applications we covered in Section 8.4 (page 145), because we will now implement the same or similar algorithms using the bidirectional BWT index simulation approach rather than the explicit suffix tree. This leads to a significant saving in space usage.

11.1.1 Maximal repeats

Recall that a maximal repeat of a string T is any substring that occurs $k > 1$ times in T and that extending the substring to the left or right in any of its occurrences yields a substring that has fewer than k occurrences. Recall also from Section 8.4.1 the simple suffix tree algorithm for finding all maximal repeats of T in linear time. Algorithm 11.1 gives a more space-efficient variant of this algorithm using the bidirectional BWT index.

We already proved that Algorithm 9.3 on page 174 visits all internal suffix tree nodes, and here we just modified the algorithm into Algorithm 11.1 to output intervals corresponding to nodes that are left- and right-maximal. From an outputted interval $[i..j]$ corresponding to a node of depth d one can reconstruct the corresponding maximal repeat in $O(d \log \sigma)$ time by extracting $T_{\mathsf{SA}[i]..\mathsf{SA}[i]+d-1}$ using Lemma 9.2. If we want to just output starting and ending positions of the maximal repeats, then we could build a succinct suffix array and use the operation `locate` to determine each starting position $\mathsf{SA}[k]$ for $k \in [i..j]$. However, this will be potentially slow, since extracting a single

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

Algorithm 11.1: Computing maximal repeats using the bidirectional BWT index.

Input: Bidirectional BWT index `idx` and interval $[1..n+1]$ corresponding to the root of suffix tree of string $T = t_1 t_2 \cdots t_n$.

Output: Suffix array intervals associated with depth information representing the maximal repeats.

```
1  Let S be an 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      if idx.isLeftMaximal(i, j) then
6          Output ([i..j], d);
7      Σ' ← idx.enumerateLeft(i, j);
8      I ← {idx.extendLeft(c, [i..j], [i'..j']) | c ∈ Σ'};
9      x ← argmax {j − i : ([i..j], [i'..j']) ∈ I};
10     I ← I \ {x};
11     ([i..j], [i'..j']) ← x;
12     if idx.isRightMaximal(i', j') then
13         S.push(x, d + 1);
14     for ([i..j], [i'..j']) ∈ I do
15         if idx.isRightMaximal(i', j') then
16             S.push(([i..j], [i'..j'], d + 1));
```

position in the suffix array takes $O(r)$ time, where r is the sampling factor used for the succinct suffix array.

In order to achieve a better extraction time, we will use the batched locate routine (Algorithm 9.2) studied in Section 9.2.4.

Suppose that the algorithm associates a maximal repeat W with an integer identifier p_W. Let $\mathsf{SA}[\overleftarrow{W}]$ denote the $|\overleftarrow{W}|$ occurrences of maximal repeat W. Recall that Algorithm 9.2 receives as input the arrays `marked` and `pairs`, the former of which marks the suffix array entries to be extracted and the latter the marked entries associated with algorithm-dependent data. To construct the input, we set the bits $\texttt{marked}[\overleftarrow{W}]$ and append $|\overleftarrow{W}|$ pairs (k, p_W) to the buffer `pairs`, for all $k \in \overleftarrow{W}$. The batched locate routine will then convert each pair (k, i) into the pair $(\mathsf{SA}[k], i)$. Then the algorithm can group all the locations of the maximal repeats by sorting `pairs` according to the maximal repeat identifiers (the second component of each pair), which can be done in optimal time using radix sort.

THEOREM 11.1 *Assume a bidirectional BWT index is built on text $T = t_1 t_2 \cdots t_n$, $t_i \in [1..\sigma]$. Algorithm 11.1 solves Problem 8.2 by outputting a representation of maximal repeats in $O(n \log \sigma + \texttt{occ})$ time and $n + O(\sigma \log^2 n)$ bits of working space, where `occ` is the output size.*

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

Proof The proof of Theorem 9.15 on page 173 applies nearly verbatim to the first phase of the algorithm. The term n corresponds to the bitvector `marked` and the term $O(\sigma \log^2 n)$ corresponds to the space used by the stack. The batched locate incurs $O(n \log \sigma + \texttt{occ})$ additional time and uses $O(\texttt{occ} \log n)$ bits of additional space, which is counted as part of the output, since the output is also of size $O(\texttt{occ} \log n)$ bits. □

We described in Section 9.3 a construction of the bidirectional BWT index in $O(n \log \sigma \log \log n)$ time using $O(n \log \sigma)$ bits of space. It was also mentioned that $O(n \log \sigma)$ time constructions exist in the literature.

11.1.2 Maximal unique matches

Recall that a maximal unique match (MUM) of two strings S and T is any substring that occurs exactly once in S and once in T and such that extending any of its two occurrences yields a substring that does not occur in the other string. Recall from Section 8.4.2 the simple suffix tree algorithm for finding all maximal unique matches in linear time. That algorithm works for multiple strings, but we will first consider the special case of two strings. Namely, Algorithm 11.2 gives a more space-efficient variant of this algorithm on two strings using the bidirectional BWT index, modifying the algorithm given above for maximal repeats.

The change to the maximal repeats algorithm consists in considering the concatenation $R = s_1 s_2 \cdots s_m \$ t_1 t_2 \cdots t_n \#$ of strings S and T, and in the addition of the indicator bitvector to check that the interval reached corresponding to a suffix tree node has a suffix of S as one child and a suffix from T as the other child. From an outputted interval $[i..i+1]$, depth d, and an indicator bitvector $I[1..n+m+2]$ one can reconstruct the corresponding maximal unique match: if, say, $I[i] = 0$, then $S_{\mathsf{SA}_R[i]..\mathsf{SA}_R[i]+d-1}$ is the maximal unique match substring, and it occurs at $\mathsf{SA}_R[i]$ in S and at $\mathsf{SA}_R[i] - m - 1$ in T.

THEOREM 11.2 *Assume a bidirectional BWT index is built on the concatenation R of strings $S = s_1 s_2 \cdots t_m$, $s_i \in [1..\sigma]$, and $T = t_1 t_2 \cdots t_n$, $t_i \in [1..\sigma]$, and a* `rank` *data structure is built on the indicator vector I such that $I[i] = 0$ iff the ith suffix of R in lexicographic order is from T, Algorithm 11.2 outputs a representation of all maximal unique matches of S and T in $O((m+n)\log \sigma)$ time and $(n+m)(2+o(1))+O(\sigma \log^2(m+n))$ bits of working space.*

Proof Again, the proof of Theorem 9.15 on page 173 applies nearly verbatim. The `rank` queries take constant time by Theorem 3.2, with a data structure occupying $o(m+n)$ bits on top of the indicator bitvector taking $m+n+2$ bits; these are counted as part of the input. □

The construction of the bidirectional BWT index in small space was discussed in the context of maximal repeats. Exercise 11.1 asks you to construct the indicator bitvector I in $O((m+n)\log \sigma)$ bits of space and in $O((m+n)\log \sigma)$ time.

Let us now consider how to extend the above approach to the general case of computing maximal unique matches on multiple strings.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

Algorithm 11.2: Computing maximal unique matches using the bidirectional BWT index

Input: Bidirectional BWT index `idx` and interval $[1..m+n+2]$ corresponding to the root of the suffix tree of string $R = s_1 s_2 \cdots s_m \$ t_1 t_2 \cdots t_n \#$, and `rank` data structure on the indicator bitvector I such that $I[i] = 0$ iff the ith suffix in the lexicographic order in R is from S.

Output: Suffix array intervals associated with depth information representing the maximal unique matches.

```
1  Let S be an empty stack;
2  S.push(([1..m+n+2], [1..m+n+2], 0));
3  while not S.isEmpty() do
4      ([i..j], [i'..j'], d) ← S.pop();
5      Scount ← rank_0(I, j) − rank_0(I, i − 1);
6      Tcount ← rank_1(I, j) − rank_1(I, i − 1);
7      if Scount < 1 or Tcount < 1 then
8          Continue;
9      if Scount = 1 and Tcount = 1 and idx.isLeftMaximal(i, j) then
10         Output ([i..j], d);
11     Σ' ← idx.enumerateLeft(i, j);
12     I ← {idx.extendLeft(c, [i..j], [i'..j']) | c ∈ Σ'};
13     x ← argmax{j − i : ([i..j], [i'..j']) ∈ I};
14     I ← I \ {x};
15     ([i..j], [i'..j']) ← x;
16     if idx.isRightMaximal(i', j') then
17         S.push(x, d + 1);
18     for ([i..j], [i'..j']) ∈ I do
19         if idx.isRightMaximal(i', j') then
20             S.push(([i..j], [i'..j'], d + 1));
```

We describe this algorithm on a more conceptual level, without detailing the adjustments to Algorithm 9.3 on a pseudocode level.

THEOREM 11.3 *Assume a bidirectional BWT index is built on the concatenation R of d strings $S^1, S^2, \ldots, S^d$ (separated by character #) of total length n, where $S_i \in [1..\sigma]^*$ for $i \in [1..d]$, then Problem 8.3 of finding all the maximal unique matches of the set of strings $\mathcal{S} = \{S^1, S^2, \ldots, S^d\}$ can be solved in time $O(n \log \sigma)$ and $O(\sigma \log^2 n) + 2n + o(n)$ bits of working space.*

Proof The maximal unique matches between the d strings can be identified by looking at intervals of BWT_R of size exactly d. The right-maximal substring that corresponds to an interval $[i..i+d-1]$ is a maximal unique match if and only if the following conditions hold.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

i. $\mathsf{BWT}_R[i..i+d-1]$ contains at least two distinct characters. This ensures left maximality.
ii. Every suffix pointed at by $\mathsf{SA}[i..i+d-1]$ belongs to a different string. This condition ensures the uniqueness of the substring in every string.

We build a bitvector of intervals $\texttt{intervals}[1..n+d]$ as follows. Initially all positions in `intervals` are set to zero. We enumerate all intervals corresponding to internal nodes of the suffix tree of R using Algorithm 9.3. For every such interval $[i..i+d-1]$ such that $\mathsf{BWT}_R[i..i+d-1]$ contains at least two distinct characters, we mark the two positions $\texttt{intervals}[i]=1$ and $\texttt{intervals}[i+d-1]=1$. This first phase allows us to identify intervals that fulfill the first condition. For each such interval we will have two corresponding bits set to one in `intervals`. We call the intervals obtained from the first phase *candidate intervals*, and we denote their number by t. In the second phase, we filter out all the candidate intervals that do not fulfill the second condition. For that purpose we allocate d bits for every candidate interval into a table $D[1..t][1..d]$ such that every position $D[i][j]$ will eventually be marked by a one if and only if one of the suffixes of string j appears in the candidate interval number i. Initially all the bits in D are set to zero. Then the marking is done as follows: we traverse the documents one by one from 1 to d, and, for each document S^j, we induce the position p in BWT_R of every suffix that appears in the document (by using *LF*-mapping) and we check whether p is inside a candidate interval. This is the case if either $\texttt{intervals}[p]=1$ or $\texttt{rank}_1(\texttt{intervals},p)$ is an odd number. If that is the case, then the number associated with the interval will be $i=\lceil \texttt{rank}_1(\texttt{intervals},p)/2\rceil$. We then set $D[i][j]=1$.

We can now scan the arrays D and `intervals` simultaneously and for every subarray $i \in [1..t]$ check whether $D[i][j]=1$ for all $j \in [1..d]$. If that is the case, then we deduce that the candidate interval number i corresponds to a maximal unique match. If this is not the case, we unmark the interval number i by setting $\texttt{intervals}[\texttt{select}_1(\texttt{intervals},2i-1)]=0$ and $\texttt{intervals}[\texttt{select}_1(\texttt{intervals},2i)]=0$.

We finally use Algorithm 9.2 to extract the positions of the MUMs in a batch. For that, we scan the bitvector `intervals` and we write a bitvector $\texttt{marked}[1..n]$ simultaneously. All the bits in `marked` are initially set to zero. During the scan of `intervals`, we write d ones in $\texttt{marked}[j..j+d-1]$ whenever we detect a MUM interval $[j..j+d-1]$ indicated by two ones at positions j and $j+d-1$ in the bitvector `intervals`, and we append the d pairs $(j,i),\ldots,(j+d-1,i)$ to buffer `pairs`, where i is the identifier of the MUM. With these inputs Algorithm 9.2 converts the first component of every pair into a text position, and then we finally group the text positions which belong to the same MUM by doing a radix sort according to the second components of the pairs. □

11.1.3 Maximal exact matches

Maximal exact matches (MEMs) are a relaxation of maximal unique matches (MUMs) in which the uniqueness property is dropped: a triple (i,j,ℓ) is a maximal exact match

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

(also called a *maximal pair*) of two strings S and T if $S_{i..i+\ell-1} = T_{j..j+\ell-1}$ and no extension of the matching substring is a match, that is, $s_{i-1} \neq t_{j-1}$ and $s_{i+\ell} \neq t_{j+\ell}$.

Problem 11.1 Maximal exact matches

Find all maximal exact matches of strings S and T.

The algorithm to solve Problem 11.1 using the bidirectional BWT index is also analogous to the one for MUMs on two strings, with some nontrivial details derived next.

Consider a left- and right-maximal interval pair $([i..j], [i'..j'])$ and depth d in the execution of Algorithm 11.2 on $R = S\$T\#$. Recall that, if $I[k] = 0$ for $i \leq k \leq j$, then $\mathsf{SA}[k]$ is a starting position of a suffix of S, otherwise $\mathsf{SA}[k] - |S| - 1$ is a starting position of a suffix of T. The cross-products of suffix starting positions from S and from T in the interval $[i..j]$ form all *candidates* for maximal pairs with $R[\mathsf{SA}[i]..\mathsf{SA}[i]+d-1]$ as the matching substring. However, not all such candidates are maximal pairs. Namely, the candidate pair (p, q) associated with $R[\mathsf{SA}[i]..\mathsf{SA}[i] + d - 1] = S[p..p + d - 1] = T[q..q + d - 1]$ is not a maximal pair if $s_{p-1} = t_{q-1}$ or $s_{p+d} = t_{q+d}$. To avoid listing candidates that are not maximal pairs, we consider next the *synchronized* traversal of two bidirectional BWT indexes, one built on S and one built on T.

Consider the interval pair $([i_0..j_0], [i'_0..j'_0])$ in the bidirectional BWT index of S corresponding to substring $R[\mathsf{SA}_R[i]..\mathsf{SA}_R[i] + d - 1]$. That is, $\mathsf{SA}_R[k] = \mathsf{SA}_S[\mathtt{rank}_0(I, k)]$ for $i \leq k \leq j$, where I is the indicator bitvector in Algorithm 11.2. Analogously, consider the interval pair $([i_1..j_1], [i'_1..j'_1])$ in the bidirectional BWT index of T corresponding to substring $R[\mathsf{SA}_R[i]..\mathsf{SA}_R[i] + d - 1]$. That is, $\mathsf{SA}_R[k] - |S| - 1 = \mathsf{SA}_T[\mathtt{rank}_1(I, k)]$ for $i \leq k \leq j$. Now, for each $a, b \in \Sigma$, we may check whether the interval pair $([i_0..j_0], [i'_0..j'_0])$ can be extended to the left with a and then to the right with b. If this is the case, $a \cdot R[\mathsf{SA}_R[i]..\mathsf{SA}_R[i] + d - 1] \cdot b$ is a substring of S. Let set A store such valid pairs a, b. Analogously, for each $c, d \in \Sigma$, we may check whether the interval pair $([i_1..j_1], [i'_1..j'_1])$ can be extended to the left with c and then to the right with d. If this is the case, $c \cdot R[\mathsf{SA}_R[i]..\mathsf{SA}_R[i]+d-1] \cdot d$ is a substring of T. Let set B store such valid pairs (c, d). We wish to compute $A \otimes B = \{(a, b, c, d) \mid (a, b) \in A, (c, d) \in B, a \neq c, b \neq d\}$, since this gives us the valid extension tuples to define the maximal pairs.

LEMMA 11.4 *Let Σ be a finite set, and let A and B be two subsets of $\Sigma \times \Sigma$. We can compute $A \otimes B = \{(a, b, c, d) \mid (a, b) \in A, (c, d) \in B, a \neq c, b \neq d\}$ in $O(|A| + |B| + |A \otimes B|)$ time.*

Proof We assume without loss of generality that $|A| < |B|$. We say that two pairs (a, b) and (c, d) are *compatible* if $a \neq c$ and $b \neq d$. Clearly, if (a, b) and (c, d) are compatible, then the only elements of $\Sigma \times \Sigma$ that are incompatible with both (a, b) and (c, d) are (a, d) and (c, b). We iteratively select a pair $(a, b) \in A$ and scan A in $O(|A|) = O(|B|)$ time to find another compatible pair (c, d): if we find one, we scan B

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

and report every pair in B that is compatible with either (a, b) or (c, d). The output will be of size $|B| - 2$ or larger, thus the time taken to scan A and B can be charged to the output. Then, we remove (a, b) and (c, d) from A and repeat the process. If A becomes empty we stop. If all the remaining pairs in A are incompatible with our selected pair (a, b), that is, if $c = a$ or $d = b$ for every $(c, d) \in A$, we build subsets A^a and A^b, where $A^a = \{(a, x) : x \neq b\} \subseteq A$ and $A^b = \{(x, b) : x \neq a\} \subseteq A$. Then we scan B, and for every pair $(x, y) \in B$ different from (a, b) we do the following. If $x \neq a$ and $y \neq b$, then we report (a, b, x, y), $\{(a, z, x, y) : (a, z) \in A^a, z \neq y\}$ and $\{(z, b, x, y) : (z, b) \in A^b, z \neq x\}$. Pairs $(a, y) \in A^a$ and $(x, b) \in A^b$ are the only ones that do not produce output, thus the cost of scanning A^a and A^b can be charged to printing the result. If $x = a$ and $y \neq b$, then we report $\{(z, b, x, y) : (z, b) \in A^b\}$. If $x \neq a$ and $y = b$, then we report $\{(a, z, x, y) : (a, z) \in A^a\}$. □

Combining Lemma 11.4 and Theorem 9.15 yields the following result.

THEOREM 11.5 *Assume that two bidirectional BWT indexes are built, one on string $S = s_1 s_2 \cdots s_m$, $s_i \in [1..\sigma]$ and one on string $T = t_1 t_2 \cdots t_n$, $t_i \in [1..\sigma]$. Algorithm 11.3 solves Problem 11.1 by outputting a representation of maximal exact matches of S and T in $O((m+n)\log\sigma + \mathtt{occ})$ time and $m+n+O(\sigma \log^2(m+n))$ bits of working space, where $\mathtt{occ}$ is the output size.*

Proof The proof of Theorem 9.15 on page 173 applies nearly verbatim to the main algorithm execution, without the conditions for left- and right-maximality, and without the calls to the cross-product subroutine.

For the first, we assume that the evaluation of conditions is executed left to right and terminated when the first condition fails; this guarantees that calls to `enumerateLeft()` and `enumerateRight()` inside conditions are returning sets of singleton elements, resulting in the claimed running time.

For the second, we need to show that the running time of the cross-product subroutine can be bounded by the input and output sizes. Indeed, each extension to the left is executed only twice, and corresponds to an explicit or implicit Weiner link in S or T. In Observation 8.14 we have shown that their amount is linear. Extensions to the right can be charged to the output. □

The construction of the bidirectional BWT index in small space was discussed in the context of maximal repeats and maximal unique matches. See Algorithms 11.3 and 11.4 for a pseudocode of the proof of Theorem 11.5. We have dropped the BWT index for the concatenation and the indicator bitvector, since the two added indexes contain all the information required in order to simulate the algorithm.

These algorithms can be used to derive all the *minimal absent words* of a single string S.

DEFINITION 11.6 *String W is a* minimal absent word *of a string $S \in \Sigma^+$ if W does not occur in S and if every proper substring of W occurs in S.*

For example, CGCGCG is a minimal absent word of string T in Figure 8.4. Therefore, to decide whether aWb is a minimal absent word of S, where $\{a, b\} \subseteq \Sigma$, it

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

suffices to check that aWb does not occur in S, and that both aW and Wb occur in S. It is easy to see that only a maximal repeat of S can be the infix W of a minimal absent word aWb (Exercise 11.5). We can thus enumerate the children of every internal node v of the suffix tree of S, then enumerate the Weiner links of v and of each one of its children, and finally compare the corresponding sets of characters in a manner similar to Algorithm 11.3. We leave the proof of the following theorem to Exercise 11.4.

Algorithm 11.3: Computing maximal exact matches using two bidirectional BWT indexes

Input: Bidirectional BWT index $\texttt{idx}_S$ and interval $[1..m+1]$ corresponding to the root of the suffix tree of string $S = s_1 s_2 \cdots s_m$, and bidirectional BWT index $\texttt{idx}_T$ and interval $[1..n+1]$ corresponding to the root of the suffix tree of string $T = t_1 t_2 \cdots t_n$.

Output: Suffix array index pairs associated with depth information representing the maximal exact matches of S and T.

Let S be an empty stack;
S.push($([1..m+1],[1..m+1],[1..n+1],[1..n+1],0)$);
while ***not*** S.isEmpty() **do**
    $([i_0..j_0],[i'_0..j'_0],[i_1..j_1],[i'_1..j'_1],\texttt{depth}) \leftarrow S$.pop();
    if $j_0 - i_0 + 1 < 1$ *or* $j_1 - i_1 + 1 < 1$ **then**
        Continue;
    if $\texttt{idx}_S$.isLeftMaximal(i_0,j_0) *or* $\texttt{idx}_T$.isLeftMaximal(i_1,j_1) *or* $\texttt{idx}_S$.enumerateLeft$(i_0,j_0) \neq \texttt{idx}_T$.enumerateLeft$(i_1,j_1)$ **then**
        Apply Algorithm 11.4 to output maximal pairs;
    $\Sigma' \leftarrow \texttt{idx}_S$.enumerateLeft$(i_0,j_0)$;
    $I \leftarrow \{(\texttt{idx}_S.\texttt{extendLeft}(c,[i_0..j_0],[i'_0..j'_0]), \texttt{idx}_T.\texttt{extendLeft}(c,[i_1..j_1],[i'_1..j'_1])) \mid c \in \Sigma'\}$;
    $x \leftarrow \text{argmax}\{\max(j_0 - i_0, j_1 - i_1) : ([i_0..j_0],[i'_0..j'_0],[i_1..j_1],[i'_1..j'_1]) \in I\}$;
    $I \leftarrow I \setminus \{x\}$;
    $([i_0..j_0],[i'_0..j'_0],[i_1..j_1],[i'_1..j'_1]) \leftarrow x$;
    if $\texttt{idx}_S$.isRightMaximal(i'_0,j'_0) *or* $\texttt{idx}_T$.isRightMaximal(i'_1,j'_1) *or* $\texttt{idx}_S$.enumerateRight$(i'_0,j'_0) \neq \texttt{idx}_T$.enumerateRight$(i'_1,j'_1)$ **then**
        S.push$(x, \texttt{depth}+1)$;
    for $([i_0..j_0],[i'_0..j'_0],[i_1..j_1],[i'_1..j'_1]) \in I$ **do**
        if $\texttt{idx}_S$.isRightMaximal(i'_0,j'_0) *or* $\texttt{idx}_T$.isRightMaximal(i'_1,j'_1) *or* $\texttt{idx}_S$.enumerateRight$(i'_0,j'_0) \neq \texttt{idx}_T$.enumerateRight$(i'_1,j'_1)$ **then**
            S.push$(([i_0..j_0],[i'_0..j'_0],[i_1..j_1],[i'_1..j'_1],\texttt{depth}+1))$;

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

Algorithm 11.4: Cross-product computation for outputting maximal exact matches using two bidirectional BWT indexes

Input: Subroutine of Algorithm 11.3 getting an interval tuple $([i_0..j_0],[i'_0..j'_0],[i_1..j_1],[i'_1..j'_1])$ and `depth` corresponding to a state of synchronized exploration of $\texttt{idx}_S$ and $\texttt{idx}_T$.

Output: Suffix array index pairs associated with depth information representing the maximal exact matches.

$A \leftarrow \emptyset$;
for $a \in \texttt{idx}_S\texttt{.enumerateLeft}(i_0, j_0)$ **do**
  $(i_a, j_a, i'_a, j'_a) \leftarrow \texttt{idx}_S\texttt{.extendLeft}(i_0, j_0, i'_0, j'_0, a)$;
  for $b \in \texttt{idx}_S\texttt{.enumerateRight}(i'_a, j'_a)$ **do**
    $A \leftarrow A \cup \{(a, b)\}$;
$B \leftarrow \emptyset$;
for $c \in \texttt{idx}_T\texttt{.enumerateLeft}(i_1, j_1)$ **do**
  $(i_c, j_c, i'_c, j'_c) \leftarrow \texttt{idx}_T\texttt{.extendLeft}(i_1, j_1, i'_1, j'_1, c)$;
  for $d \in \texttt{idx}_T\texttt{.enumerateRight}(i'_c, j'_c)$ **do**
    $B \leftarrow B \cup \{(c, d)\}$;
for $(a, b, c, d) \in A \otimes B$ **do**
  $(i_a, j_a, i'_a, j'_a) \leftarrow \texttt{idx}_S\texttt{.extendLeft}(i_0, j_0, i'_0, j'_0, a)$;
  $(i_b, j_b, i'_b, j'_b) \leftarrow \texttt{idx}_S\texttt{.extendRight}(i_a, j_a, i'_a, j'_a, b)$;
  $(i_c, j_c, i'_c, j'_c) \leftarrow \texttt{idx}_T\texttt{.extendLeft}(i_1, j_1, i'_1, j'_1, c)$;
  $(i_d, j_d, i'_d, j'_d) \leftarrow \texttt{idx}_T\texttt{.extendRight}(i_c, j_c, i'_c, j'_c, d)$;
  for $i \in [i_b..j_b]$ **do**
    for $j \in [i_d..j_d]$ **do**
      Output $(i, j, \texttt{depth})$;

THEOREM 11.7 *Assume that the bidirectional BWT index is built on a string $S \in [1..\sigma]^n$. All the minimal absent words of S can be computed in $O(n \log \sigma + \texttt{occ})$ time, and in $O(\sigma \log^2 n)$ bits of working space, where* `occ` *is the output size.*

Note that `occ` can be of size $\Theta(n\sigma)$ in this case: see Exercises 11.6 and 11.7.

11.2 Comparing genomes without alignment

Given two strings S and T, a *string kernel* is a function that simultaneously converts S and T into vectors $\{\mathbf{S}, \mathbf{T}\} \subset \mathbb{R}^n$ for some $n > 0$, and computes a similarity or a distance measure between $\mathbf{S}$ and $\mathbf{T}$, *without building and storing* $\mathbf{S}$ *and* $\mathbf{T}$ *explicitly*. The (possibly infinite) dimensions of the resulting vectors are typically all substructures of a specific type, for example all strings of a fixed length on the alphabet of S and T, and the value assigned to vectors $\mathbf{S}$ and $\mathbf{T}$ along dimension W corresponds to the frequency

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

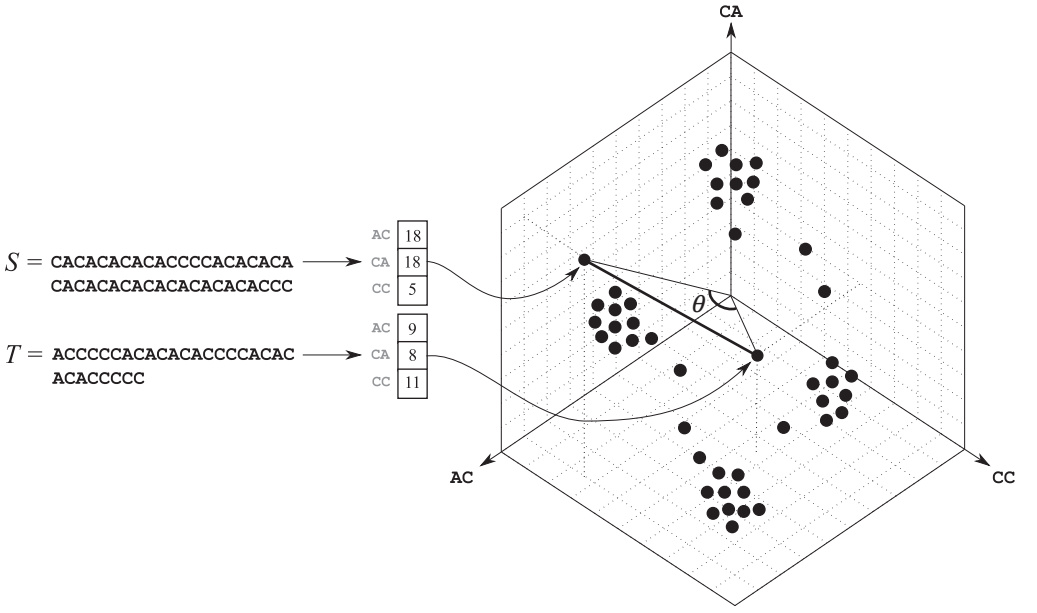

Figure 11.1 The k-mer kernel $\kappa(S, T)$ for $k = 2$, which coincides with the cosine of the angle θ shown. The bold segment is the norm $||\mathbf{S} - \mathbf{T}||_2$, or equivalently the Euclidean distance between $\mathbf{S}$ and $\mathbf{T}$ in $\mathbb{R}^3$.

of substructure W in strings S and T, respectively – often rescaled and corrected in domain-specific ways (see the example in Figure 11.1). For this reason, $\mathbf{S}$ and $\mathbf{T}$ are called *composition vectors*. A large number of components in $\mathbf{S}$ and $\mathbf{T}$ can be zero in practice: in this case, we say that $\mathbf{S}$ and $\mathbf{T}$ are *sparse*. The term *kernel* derives from learning theory: see Insight 11.2 for a general definition of kernels on vectors.

Conceptually, comparing genomes using the frequency of their substructures captures genome-wide *compositional biases* that are well known to correlate with accepted species classifications in biology. It thus allows one to estimate the similarity between two evolutionary distant species, for which a whole-genome alignment would not be meaningful, if a trace of their evolutionary history is still preserved by their compositional preferences. Composition-based kernels extend naturally to sets of reads and to metagenomic samples (see Section 16.3), estimating their similarity without the need for assembly, and they can be generalized to mixed settings in which sets of reads must be compared with sequenced genomes, and vice versa. Computationally, string kernels tend to be more time- and space-efficient than whole-genome alignments, and are thus the ideal candidates for applications that must estimate the similarity of two long strings very quickly, or that need to compute a similarity measure for all pairs of strings in a massive dataset.

This section walks the reader through a number of algorithms that compute string kernels, in roughly increasing order of expressive power and algorithmic sophistication. Most such algorithms can be implemented using suffix trees, and some of them can be mapped to the space-efficient genome analysis techniques described in Section 11.1: see

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

Table 11.1 Overview of the kernels between strings S and T presented in Section 11.2 and implemented using suffix trees. Each kernel is mapped to the data structures and algorithms used to implement it. Numbers indicate sections that describe a kernel explicitly. Bullets indicate solutions that we mention but do not describe explicitly. GST: generalized suffix tree of S and T. MS: suffix tree of T and matching statistics of S with respect to T. BWT: bidirectional BWT indexes of S and of T.

	GST	MS	BWT
k-mer kernel	•	11.2.3	11.2.1
Substring kernel	•	11.2.3	11.2.1
Generalized substring kernel	•	11.2.3	
Markovian substring kernel	11.2.2		•
Variable-memory Markov chain		11.2.3	

Table 11.1 for an overview. As will be apparent in what follows, the key to efficiency lies in the beautiful interplay between the probabilities of substrings and the combinatorial properties of suffix trees.

We will repeatedly call S and T the two strings on alphabet $\Sigma = [1..\sigma]$ which we want to compare, and we will call *k-mer* any string $W \in [1..\sigma]$ of fixed length $k > 0$. We will also denote by $f_S(W)$ the number of (possibly overlapping) occurrences of string W in S, using the shorthand $p_S(W) = f_S(W)/(|S| - k + 1)$ to denote the *empirical probability* of observing W in S. As is customary, we will denote the suffix tree of S by ST_S, dropping the subscript whenever it is implicit from the context.

Throughout this section we will be mostly interested in computing the *cosine of the angle between the composition vectors* $\mathbf{S}$ *and* $\mathbf{T}$, a natural measure of their similarity. In other words, the kernels we will consider have the form

$$\kappa(\mathbf{S}, \mathbf{T}) = \frac{\sum_W \mathbf{S}[W]\mathbf{T}[W]}{\sqrt{\left(\sum_W \mathbf{S}[W]^2\right)\left(\sum_W \mathbf{T}[W]^2\right)}}. \tag{11.1}$$

Exercise 11.16 gives some intuition on why $\kappa(\mathbf{S}, \mathbf{T})$ measures the cosine of the angle between $\mathbf{S}$ and $\mathbf{T}$. Note that $\kappa(\mathbf{S}, \mathbf{T})$, a measure of similarity in $[-1..1]$ that equals zero when $\mathbf{S}$ and $\mathbf{T}$ have no substructure in common, can be converted into a *distance* $d(\mathbf{S}, \mathbf{T}) = (1 - \kappa(\mathbf{S}, \mathbf{T}))/2$ that falls within the range $[0..1]$. Most of the algorithms described in this section apply to other commonly used norms of vector $\mathbf{S} - \mathbf{T}$, like the p-norm,

$$||\mathbf{S} - \mathbf{T}||_p = \left(\sum_W |\mathbf{S}[W] - \mathbf{T}[W]|^p\right)^{1/p},$$

and the infinity norm,

$$||\mathbf{S} - \mathbf{T}||_\infty = \max_W \{|\mathbf{S}[W] - \mathbf{T}[W]|\}.$$

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

Even more generally, most algorithms in this section apply to any function of the three arguments N, D_S, and D_T defined as follows:

$$N = \bigoplus_W g_1(\mathbf{S}[W], \mathbf{T}[W]),$$
$$D_S = \bigotimes_W g_2(\mathbf{S}[W]),$$
$$D_T = \bigodot_W g_3(\mathbf{T}[W]),$$

where g_1, g_2, and g_3 are arbitrary functions, and $\oplus$, $\otimes$, and $\odot$ are arbitrary *commutative* and *associative* binary operations, or equivalently operations (like sum and product) whose result depends neither on the order of the operands nor on the order in which the operations are performed on such operands.

11.2.1 Substring and k-mer kernels

Perhaps the simplest way to represent a string as a composition vector consists in counting the frequency of all its distinct substrings of a fixed length. This is known as the *k-mer spectrum* of the string.

DEFINITION 11.8 *Given a string $S \in [1..\sigma]^+$ and a length $k > 0$, let vector $\mathbf{S}_k = [1..\sigma^k]$ be such that $\mathbf{S}_k[W] = f_S(W)$ for every $W \in [1..\sigma]^k$. The* k-mer complexity *$C(S,k)$ of string S is the number of non-zero components of $\mathbf{S}_k$. The* k-mer kernel *between two strings S and T is Equation (11.1) evaluated on $\mathbf{S}_k$ and $\mathbf{T}_k$.*

Put differently, the k-mer complexity of S is the number of distinct k-mers that occur in S. Choosing the best value of k for a specific application might be challenging: Insight 11.3 describes a possible data-driven strategy. Alternatively, we could just remove the constraint of a fixed k, and use the more general notions of the *substring complexity* and the *substring kernel*.

Insight 11.2 Kernels on vectors

Given a set of vectors $\mathcal{S} \subset \mathbb{R}^n$ such that each vector belongs to one of two possible classes, assume that we have a *classification algorithm* $\mathcal{A}$ that divides the two classes with a plane in $\mathbb{R}^n$. If we are given a new vector, we can label it with the identifier of a class by computing the side of the plane it lies in. If the two classes cannot be separated by a plane, or, equivalently, if they are not *linearly separable* in $\mathbb{R}^n$, we could create a *nonlinear map* $\phi : \mathbb{R}^n \mapsto \mathbb{R}^m$ with $m > n$ such that they become linearly separable in $\mathbb{R}^m$, and then run algorithm $\mathcal{A}$ on the projected set of points $\mathcal{S}' = \{\phi(\mathbf{X}) \mid \mathbf{X} \in \mathcal{S}\}$. This is an ingenious way of building a *nonlinear* classification algorithm in $\mathbb{R}^n$ using a *linear* classification algorithm in $\mathbb{R}^m$.

Assume that algorithm $\mathcal{A}$ uses just inner products $\mathbf{X} \cdot \mathbf{Y}$ between pairs of vectors $\{\mathbf{X}, \mathbf{Y}\} \subseteq \mathcal{S}$. Building and storing the projected m-dimensional vectors, and computing inner products between pairs of such vectors, could be significantly

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

more expensive than computing inner products between the original n-dimensional vectors. However, if ϕ is such that $\phi(\mathbf{X}) \cdot \phi(\mathbf{Y}) = \kappa(\mathbf{X}, \mathbf{Y})$ for some symmetric function κ that is fast to evaluate, we could run $\mathcal{A}$ on $\mathcal{S}'$ *without constructing* $\mathcal{S}'$ *explicitly*, by just replacing all calls to inner products by calls to function κ in the code of $\mathcal{A}$: this approach is called *kernelizing* algorithm $\mathcal{A}$, and κ is called a *kernel function*. The family of functions $\kappa : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}$ for which there is a mapping ϕ such that $\kappa(\mathbf{X}, \mathbf{Y}) = \phi(\mathbf{X}) \cdot \phi(\mathbf{Y})$ is defined by the so-called *Mercer's conditions*. The symmetric, positive-definite matrix containing $\kappa(\mathbf{X}, \mathbf{Y})$ for all $\{\mathbf{X}, \mathbf{Y}\} \subset \mathcal{S}$ is called the *Gram matrix* of $\mathcal{S}$. Multiple kernel functions can be combined to obtain more complex kernels.

If the input set $\mathcal{S}$ contains objects other than vectors, kernelizing $\mathcal{A}$ means replacing all the inner products $\mathbf{X} \cdot \mathbf{Y}$ in $\mathcal{A}$ by calls to a function $\kappa(X, Y)$ that simultaneously converts objects $\{X, Y\} \in \mathcal{S}$ into vectors $\{\mathbf{X}, \mathbf{Y}\} \subset \mathbb{R}^m$ and computes the inner product $\mathbf{X} \cdot \mathbf{Y}$, without building and storing $\mathbf{X}$ and $\mathbf{Y}$ explicitly.

DEFINITION 11.9 *Given a string $S \in [1..\sigma]^+$, consider the infinite-dimensional vector $\mathbf{S}_\infty$, indexed by all distinct substrings $W \in [1..\sigma]^+$, such that $\mathbf{S}_\infty[W] = f_S(W)$. The* substring complexity $C(S)$ *of string S is the number of non-zero components of $\mathbf{S}_\infty$. The* substring kernel *between two strings S and T is Equation (11.1) evaluated on $\mathbf{S}_\infty$ and $\mathbf{T}_\infty$.*

Computing substring complexity and substring kernels, with or without a constraint on string length, is trivial using suffix trees: see Exercise 11.17. In this section we show how to perform the same computations space-efficiently, using the bidirectional BWT index described in Section 9.4. Recall that Algorithm 9.3 enumerates all internal nodes of the suffix tree of S using the bidirectional BWT index of S and a stack of bounded size. Recall also that, to limit the size of this stack, the nodes of ST_S are enumerated *in a specific order*, related to the topology of the suffix-link tree of S. In what follows, we describe algorithms to compute substring complexity and substring kernels *in a way that does not depend on the order in which the nodes of* ST_S *are enumerated*: we can thus implement such algorithms on top of Algorithm 9.3.

We first describe the simpler cases of k-mer complexity and substring complexity, and then we extend their solutions to kernels. Similar algorithms can compute the *kth-order empirical entropy* of S: see Exercise 11.13. The main idea behind all such algorithms consists in a *telescoping approach* that works by adding and subtracting terms in a sum, as described in the next lemma and visualized in Figure 11.2.

LEMMA 11.10 *Given the bidirectional BWT index of a string $S \in [1..\sigma]^n$ and an integer k, there is an algorithm that computes the k-mer complexity $C(S, k)$ of S in $O(n \log \sigma)$ time and $O(\sigma \log^2 n)$ bits of working space.*

Proof A k-mer of S can either be the label of a node of ST_S, or it could end in the middle of an edge (u, v) of the suffix tree. In the latter case, we assume that the k-mer is represented by its locus v, which might be a leaf. Let $C(S, k)$ be initialized to $|S| + 1 - k$, that is, to the number of leaves that correspond to suffixes of $S\#$ of length

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

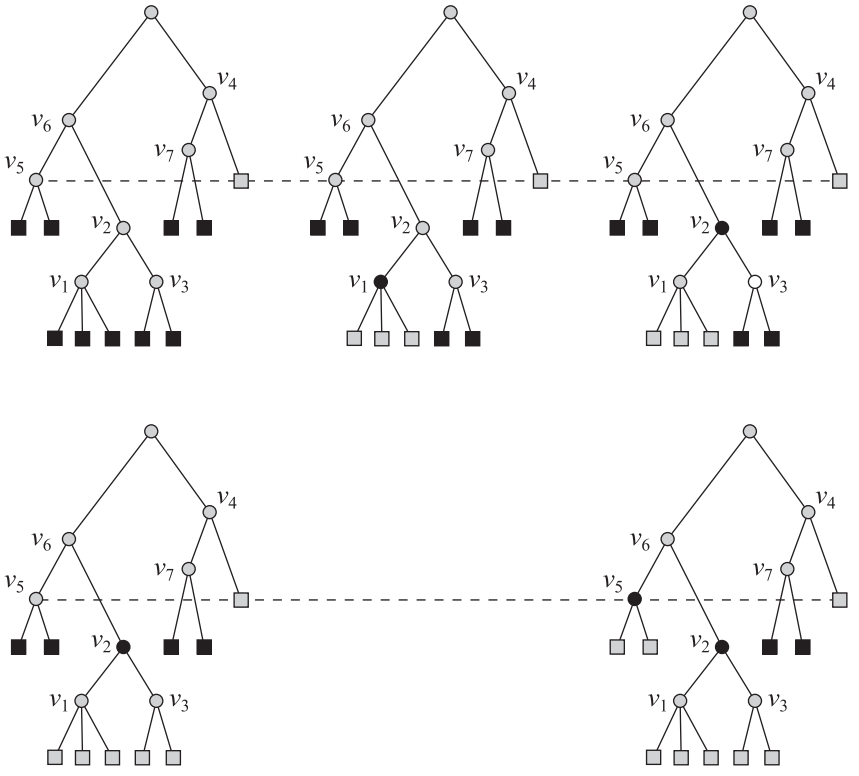

Figure 11.2 Lemma 11.10 illustrated from top to bottom, from left to right, on the suffix tree of string S, assuming that its internal nodes are enumerated in the order $v_1, v_2, \ldots, v_7$. Black indicates a $+1$ contribution to $C(S,k)$, gray indicates zero contribution, and white indicates a -1 contribution. Note that v_3 is white at step 3. The dashed line corresponds to k. The empty space without a tree at step 5 indicates that nothing happens when node v_4 is enumerated.

at least k, excluding suffix $S[|S|-k+2..|S|]\#$. We use Algorithm 9.3 to enumerate the internal nodes of ST, and every time we enumerate a node v we proceed as follows. If $|\ell(v)| < k$ we leave $C(S,k)$ unaltered, otherwise we increment $C(S,k)$ by one and decrement $C(S,k)$ by the number of children of v in ST. This number is the size of the output of the operation `enumerateRight` provided by the bidirectional BWT index of S, and the value $|\ell(v)|$ is provided by Algorithm 9.3. It follows that every node v of ST that is located at depth at least k and that is not the locus of a k-mer is both added to $C(S,k)$ (when the algorithm visits v) and subtracted from $C(S,k)$ (when the algorithm visits $\texttt{parent}(v)$). Leaves at depth at least k are added by the initialization of $C(S,k)$, and subtracted during the enumeration. Conversely, every locus v of a k-mer of S (including leaves) is just added to $C(S,k)$, because $|\ell(\texttt{parent}(v))| < k$. □

Computing the k-mer kernel between two strings S and T follows the same telescoping strategy.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

LEMMA 11.11 *Given an integer k and the bidirectional BWT index of a string $S \in [1..\sigma]^n$ and of a string $T \in [1..\sigma]^m$, there is an algorithm that computes the k-mer kernel between S and T in $O((n+m)\log\sigma)$ time and $O(\sigma \log^2(m+n))$ bits of working space.*

Proof Recall that Algorithm 11.3 in Section 11.1.3 enumerates all the internal nodes of the generalized suffix tree of S and T, using the bidirectional BWT index of S, the bidirectional BWT index of T, and a stack of bounded size. We compute the substring kernel by enumerating the internal nodes of the generalized suffix tree of S and T as done in Algorithm 11.3, and by simultaneously updating three variables, denoted by N, D_S, and D_T, which assume the following values after all the internal nodes of the generalized suffix tree have been enumerated:

$$N = \sum_{W \in [1..\sigma]^k} \mathbf{S}_k[W]\mathbf{T}_k[W],$$
$$D_S = \sum_{W \in [1..\sigma]^k} \mathbf{S}_k[W]^2,$$
$$D_T = \sum_{W \in [1..\sigma]^k} \mathbf{T}_k[W]^2.$$

Thus, after the enumeration, the value of the k-mer kernel is $N/\sqrt{D_S \cdot D_T}$. We initialize the variables as follows: $N = 0$, $D_S = |S| + 1 - k$, and $D_T = |T| + 1 - k$. These are the contributions of the $|S| + 1 - k + |T| + 1 - k$ leaves at depth at least k in the generalized suffix tree of S and T, excluding the leaves which correspond to suffixes $S[|S| - k + 2..|S|]\#$ and $T[|T| - k + 2..|T|]\#$. Whenever Algorithm 11.3 enumerates a node u of the generalized suffix tree of S and T, where $\ell(u) = W$, we keep all variables unchanged if $|W| < k$. Otherwise, we update them as follows:

$$N \leftarrow N + f_S(W)f_T(W) - \sum_{v \in \texttt{children}(u)} f_S\big(\ell(v)\big) \cdot f_T\big(\ell(v)\big),$$
$$D_S \leftarrow D_S + f_S(W)^2 - \sum_{v \in \texttt{children}(u)} f_S\big(\ell(v)\big)^2,$$
$$D_T \leftarrow D_T + f_T(W)^2 - \sum_{v \in \texttt{children}(u)} f_T\big(\ell(v)\big)^2,$$

where the values $f_S(W)$ and $f_T(W)$ are the sizes of the intervals which represent node u in the two BWT indexes, the values $f_S\big(\ell(v)\big)$ and $f_T\big(\ell(v)\big)$ can be computed for every child v of u using the operations `enumerateRight` and `extendRight` provided by the bidirectional BWT index of S and T, and the value $|\ell(v)|$ is provided by Algorithm 11.3. In analogy to Lemma 11.10, the contribution of the loci of the distinct k-mers of S, of T, or of both, is added to the three variables and never subtracted, while the contribution of every other node v at depth at least k in the generalized suffix tree of S and T (including leaves) is both added (when the algorithm visits v, or when N, D_S, and D_T are initialized) and subtracted (when the algorithm visits $\texttt{parent}(v)$). □

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

This algorithm can easily be adapted to compute the *Jaccard distance* between the set of k-mers of S and the set of k-mers of T, and to compute a variant of the k-mer kernel in which $\mathbf{S}_k[W] = p_S(W)$ and $\mathbf{T}_k[W] = p_T(W)$ for every $W \in [1..\sigma]^k$: see Exercises 11.10 and 11.11. Computing the substring complexity and the substring kernel amounts to applying the same telescoping strategy, but with different contributions.

COROLLARY 11.12 *Given the bidirectional BWT index of a string $S \in [1..\sigma]^n$, there is an algorithm that computes the substring complexity $C(S)$ in $O(n \log \sigma)$ time and $O(\sigma \log^2 n)$ bits of working space.*

Proof Note that the substring complexity of S coincides with the number of characters in $[1..\sigma]$ that occur on all edges of ST_S. We can thus proceed as in Lemma 11.10, initializing $C(S)$ to $n(n+1)/2$, or equivalently to the sum of the lengths of all suffixes of S. Whenever we visit a node u of ST_S, we add to $C(S)$ the quantity $|\ell(u)|$, and we subtract from $C(S)$ the quantity $|\ell(u)| \cdot |\texttt{children}(v)|$. The net effect of all such operations coincides with summing the lengths of all edges of ST, discarding all occurrences of character # (see Figure 11.3). Note that $|\ell(u)|$ is provided by Algorithm 9.3, and $|\texttt{children}(v)|$ is the size of the output of operation `enumerateRight` provided by the bidirectional BWT index of S. □

COROLLARY 11.13 *Given the bidirectional BWT index of a string $S \in [1..\sigma]^n$ and of a string $T \in [1..\sigma]^m$, there is an algorithm that computes the substring kernel between S and T in $O((n+m)\log\sigma)$ time and $O(\sigma \log^2(m+n))$ bits of working space.*

Proof We proceed as in Lemma 11.11, maintaining the variables N, D_S, and D_T, which assume the following values after all the internal nodes of the generalized suffix tree of S and T have been enumerated:

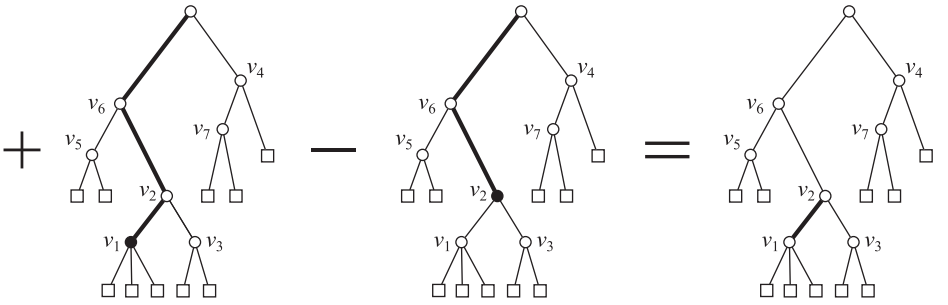

Figure 11.3 Illustrating Corollary 11.12 on the suffix tree of string S. When the algorithm enumerates v_1, it adds to $C(S)$ the number of characters on the path from the root to v_1. When the algorithm enumerates v_2, it subtracts the number of characters on the path from the root to v_2. The net contribution of these two operations to $C(S)$ is the number of characters on edge (v_2, v_1).

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

$$N = \sum_{W \in [1..\sigma]^+} \mathbf{S}_\infty[W]\mathbf{T}_\infty[W],$$
$$D_S = \sum_{W \in [1..\sigma]^+} \mathbf{S}_\infty[W]^2,$$
$$D_T = \sum_{W \in [1..\sigma]^+} \mathbf{T}_\infty[W]^2.$$

Thus, after the enumeration, the value of the k-mer kernel is $N/\sqrt{D_S \cdot D_T}$. We initialize such variables as follows: $N = 0$, $D_S = n(n+1)/2$, and $D_T = m(m+1)/2$. When we visit a node u of the generalized suffix tree of S and T, with $\ell(u) = W$, we update the variables as follows:

$$N \leftarrow N + |W| \cdot \left(f_S(W) f_T(W) - \sum_{v \in \texttt{children}(u)} f_S\big(\ell(v)\big) \cdot f_T\big(\ell(v)\big) \right),$$
$$D_S \leftarrow D_S + |W| \cdot \left(f_S(W)^2 - \sum_{v \in \texttt{children}(u)} f_S\big(\ell(v)\big)^2 \right),$$
$$D_T \leftarrow D_T + |W| \cdot \left(f_T(W)^2 - \sum_{v \in \texttt{children}(u)} f_T\big(\ell(v)\big)^2 \right).$$

□

The equations in the proof of Corollary 11.13 can be extended to compute a variant of the substring kernel in which $\mathbf{S}_\infty[W] = p_S(W)$ and $\mathbf{T}_\infty[W] = p_T(W)$ for every $W \in [1..\sigma]^k$: see Exercise 11.12. The telescoping technique described in this section can also be used to compute kernels and complexity measures that are restricted to substrings whose length belongs to a user-specified range.

Insight 11.3 Choosing k from the data

Given a string $S \in [1..\sigma]^n$, let $g : [1..n-1] \mapsto [1..\lfloor n/2 \rfloor]$ be a map from the length ℓ of a substring of S to the number of distinct ℓ-mers that occur *at least twice* in S. In many genome-scale applications, k is set to be at least $k_1 = \text{argmax}\{g(\ell) \mid \ell \in [1..n-1]\}$. This value, which is close to $\log_\sigma n$ for strings generated by independent, identically distributed random sources, can be derived exactly from S, by adapting the algorithm described in Lemma 11.10: see Exercise 11.15.

To determine an *upper bound* on k from the data, recall from Section 11.2.2 that, if S was generated by a Markov process of order $\ell - 2$ or smaller, the expected relative frequency of an ℓ-mer W in S can be approximated by Equation (11.3):

$$\tilde{p}_S(W) = \frac{p_S(W[1..\ell-1]) \cdot p_S(W[2..\ell])}{p_S(W[2..\ell-1])}.$$

It makes sense to disregard values of ℓ for which the probability distributions induced by p_S (the observed relative frequency) and by $\tilde{p}_S$ (the expected relative frequency)

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

over all ℓ-mers are very similar. More formally, let $\mathbf{S}_\ell[1..\sigma^\ell]$ be the composition vector of S indexed by all possible ℓ-mers on alphabet $\Sigma = [1..\sigma]$, such that $\mathbf{S}_\ell[W] = p_S(W)$ for all $W \in [1..\sigma]^\ell$. Let $\tilde{\mathbf{S}}_\ell[1..\sigma^\ell]$ be the normalized composition vector of the estimates $\tilde{p}_S$, or equivalently $\tilde{\mathbf{S}}_\ell[W] = \tilde{p}_S(W) / \sum_{W \in [1..\sigma]^\ell} \tilde{p}_S(W)$ for all $W \in [1..\sigma]^\ell$. Recall from Section 6.4 that

$$\mathsf{KL}(\mathbf{S}_\ell, \tilde{\mathbf{S}}_\ell) = \sum_{W \in [1..\sigma]^\ell} \mathbf{S}_\ell[W] \cdot \Big(\log(\mathbf{S}_\ell[W]) - \log(\tilde{\mathbf{S}}_\ell[W]) \Big)$$

is the Kullback–Leibler divergence of probability distribution $\tilde{\mathbf{S}}_\ell$ from probability distribution $\mathbf{S}_\ell$. Thus, let $Q(\ell) = \sum_{i=\ell}^{\infty} \mathsf{KL}(\mathbf{S}_\ell, \tilde{\mathbf{S}}_\ell)$. If $Q(\ell) \approx 0$ for some ℓ, then vector $\mathbf{S}_k$ can be derived almost exactly from vectors $\mathbf{S}_{k-1}$ and $\mathbf{S}_{k-2}$ for all $k \geq \ell$. Therefore, we can disregard values of k larger than $k_2 = \min\{\ell \mid Q(\ell) \approx 0\}$.

Given a collection of strings $\mathcal{S} = \{S^1, S^2, \ldots, S^m\}$, k should belong to the intersection of all intervals $[k_1^i..k_2^i]$ induced by strings $S^i \in \mathcal{S}$. However, note that both k_1^i and k_2^i depend on the length of S^i, and the intersection of all intervals might be empty. Without loss of generality, assume that S^1 is the shortest string in $\mathcal{S}$. A practical solution to this problem consists in partitioning every string in $\mathcal{S} \setminus \{S^1\}$ into blocks of size $|S^1|$, and in deciding the value of k by intersecting the intervals induced by all such blocks. Let $d : \mathcal{S} \times \mathcal{S} \mapsto \mathbb{R}$ be any k-mer distance between two strings in $\mathcal{S}$, and consider strings $S^i = S_1^i \cdot S_2^i \cdot \cdots \cdot S_p^i$ and $S^j = S_1^j \cdot S_2^j \cdots \cdots S_q^j$, where S_x^i and S_y^j are blocks of size $|S^1|$. We can replace $d(S^i, S^j)$ in practice by

$$d'(S^i, S^j) = \frac{1}{2} \left(\frac{1}{p} \sum_{a=1}^{p} \min \left\{ d(S_a^i, S_b^j) \mid b \in [1..q] \right\} + \frac{1}{q} \sum_{b=1}^{q} \min \left\{ d(S_b^j, S_a^i) \mid a \in [1..p] \right\} \right).$$

*11.2.2 Substring kernels with Markovian correction

In some applications it is desirable to assign to component $W \in [1..\sigma]^+$ of composition vector $\mathbf{S}$ a score that measures the *statistical significance* of observing $f_S(W)$ occurrences of substring W in string $S \in [1..\sigma]^n$, rather than just the frequency $f_S(W)$ or the relative frequency $p_S(W)$. A string whose frequency departs from its expected value is more likely to carry biological signals, and kernels that take statistical significance into account are indeed more accurate when applied to genomes of evolutionarily distant species. In this section we focus on a score that is particularly effective in measuring the similarity of genomes.

We start by assuming that the input strings are generated by a Markov random process (see Chapter 7), which is a realistic assumption for genomes.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

LEMMA 11.14 *Consider a Markov random process of order $k-2$ or smaller that generates strings on alphabet $\Sigma = [1..\sigma]$ according to a probability distribution $\mathbb{P}$. Let $W \in [1..\sigma]^k$. The probability of observing W in a string generated by the random process is*

$$\mathbb{P}(W) = \frac{\mathbb{P}(W[1..k-1]) \cdot \mathbb{P}(W[2..k])}{\mathbb{P}(W[2..k-1])}. \tag{11.2}$$

Proof The probability of observing W in a string generated by the random process is

$$\begin{aligned}\mathbb{P}(W) &= \mathbb{P}(W[1..k-1]) \cdot \mathbb{P}(W[k] \mid W[1..k-1]) \\ &= \mathbb{P}(W[1..k-1]) \cdot \mathbb{P}(W[k] \mid W[2..k-1]),\end{aligned}$$

where the second equality comes from the fact that the process is Markovian with order $k-2$ or smaller. The claim follows from combining this equation with

$$\mathbb{P}(W[2..k]) = \mathbb{P}(W[k] \mid W[2..k-1]) \cdot \mathbb{P}(W[2..k-1]). \qquad \square$$

We can estimate $\mathbb{P}(W)$ with the empirical probability $p_S(W)$, which might be zero for some W, obtaining the following approximation of Equation (11.2):

$$\tilde{p}_S(W) = \begin{cases} p_S(W[1..k-1]) \cdot p_S(W[2\ldots k]) \,/\, p_S(W[2\ldots k-1]) & \text{if } p_S(W[2\ldots k-1]) \neq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{11.3}$$

Other estimators for the expected probability of a k-mer exist, with varying practical effectiveness for classifying genomes. We focus here on Equation (11.3), both because of its natural formulation in terms of Markov processes, and because it maximizes a notion of entropy detailed in Insight 11.4. We measure the significance of the event that substring W has empirical probability $p_S(W)$ in string S by the score

$$z_S(W) = \begin{cases} \big(p_S(W) - \tilde{p}_S(W)\big)/\tilde{p}_S(W) & \text{if } \tilde{p}_S(W) \neq 0, \\ 0 & \text{if } \tilde{p}_S(W) = 0. \end{cases}$$

Note that, even though W does not occur in S, $z_S(W)$ can be -1 if both $f_S(W[1..k-1]) > 0$ and $f_S(W[2..k]) > 0$. Recall from Section 11.1.3 that such a W is a *minimal absent word* of S, and that there is an algorithm to compute all the minimal absent words of S in $O(n\sigma)$ time and $O(n(1+o(1))+\sigma\log^2 n)$ bits of working space, using the bidirectional BWT index of S (see Theorem 11.7).

After elementary manipulations, the formula for $z_S(W)$ becomes

$$\begin{aligned} z_S(W) &= g(n,k) \cdot \frac{h_S(W[1..k])}{h_S(W[2..k])} - 1, \\ h_S(X) &= \frac{f_S(X)}{f_S(X[1..|X|-1])}, \\ g(n,k) &= \frac{(n-k+2)^2}{(n-k+1)(n-k+3)}. \end{aligned}$$

Let $\mathbf{S}_z$ be an infinite vector indexed by *all strings* in $[1..\sigma]^+$, and such that $\mathbf{S}_z[W] = z_S(W)$. As before, we want to compute Equation (11.1) on the composition vectors $\mathbf{S}_z$

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

and $\mathbf{T}_z$ of two input strings $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$. To do so, we iterate over all substrings of S and T that contribute to Equation (11.1), including minimal absent words. For clarity, we describe our algorithms on the generalized suffix tree ST_{ST} of S and T, leaving to Exercise 11.18 the task of implementing them on the space-efficient enumeration of the internal nodes of ST_{ST} described in Algorithm 11.3.

Assume that ST_{ST} is augmented with explicit and implicit Weiner links, and that every destination of an implicit Weiner link is assigned a corresponding unary node in ST. We denote by ST^* this augmented generalized suffix tree. Recall from Observation 8.14 that the size of ST^* is still $O(n+m)$. Let $\texttt{parent}(v)$ denote the (possibly unary) parent of a node v in ST^*, let $\texttt{child}(v,c)$ denote the (possibly unary) child of node v reachable from an edge whose label starts by character $c \in [1..\sigma]$, and let $\texttt{suffixLink}(v)$ denote the node that is reachable by taking the suffix link from v, or equivalently by following in the opposite direction the Weiner link that ends at node v. We organize the computation of Equation (11.1) in three phases, corresponding to the contributions of three different types of string.

In the first phase, we consider all strings that label the internal nodes and the leaves of ST^*.

Insight 11.4 Maximum-entropy estimators of k-mer probability

Let W be a $(k-1)$-mer on alphabet $[1..\sigma]$. To simplify the notation, let $W = xVy$, where x and y are characters in $[1..\sigma]$ and $V \in [1..\sigma]^*$. Consider the estimator of the probability of W in Equation (11.3). Note that

$$\sum_{a\in[1..\sigma]} \tilde{p}_S(Wa) = \frac{p_S(W)}{p_S(Vy)} \cdot \sum_{a\in[1..\sigma]} p_S(Vya),$$

$$\sum_{a\in[1..\sigma]} \tilde{p}_S(aW) = \frac{p_S(W)}{p_S(xV)} \cdot \sum_{a\in[1..\sigma]} p_S(axV),$$

where the right-hand sides of these equations are constants that can be computed from counts of substrings in S. By applying these equations to every $(k-1)$-mer W we get a system of $2\sigma^{k-1}$ constraints on function $\tilde{p}(S)$ evaluated at σ^k points, and the system is under-determined if $\sigma > 2$. Therefore, Equation (11.3) can be interpreted as *just one of the many possible functions that satisfy all such constraints*. Inside this set of feasible solutions, it is common practice to choose the one that maximizes the entropy $-\sum_{U\in[1..\sigma]^k} \tilde{p}_S(U)\log \tilde{p}_S(U)$, or equivalently the one that makes the weakest possible assumptions on the function beyond the constraints themselves. It is easy to see that this optimization problem, with the additional constraint that $\tilde{p}_S(U) \geq 0$ for all $U \in [1..\sigma]^k$, can be decoupled into σ^{k-2} subproblems that can be solved independently. Surprisingly, it turns out that Equation (11.3) already maximizes the entropy. However, this is not true for any function. For example, consider another popular estimator,

$$\tilde{p}_S(W) = \frac{p_S(x)p_S(Vy) + p_S(xV)p_S(y)}{2},$$

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

derived under the assumption that $k \geq 2$ and that x, V, and y occur independently in S. The maximum-entropy version of this function is

$$\tilde{p}_S(W) = \frac{\big(p_S(xV) + p_S(x)\alpha\big) \cdot \big(p_S(Vy) + p_S(y)\beta\big)}{2(\alpha + \beta)}$$

where $\alpha = \sum_{a \in [1..\sigma]} p_S(Va)$ and $\beta = \sum_{a \in [1..\sigma]} p_S(aV)$.

LEMMA 11.15 *Given* ST^**, the values of* $z_S(W)$ *and of* $z_T(W)$ *for all strings* W *such that* $W = \ell(v)$ *for some node* v *of* ST^* *can be computed in overall* $O(n+m)$ *time.*

Proof We store in every node v of ST^* two integers, $\texttt{fs} = f_S\big(\ell(v)\big)$ and $\texttt{ft} = f_T\big(\ell(v)\big)$, where $\ell(v)$ is the label of the path from the root to v in ST^*. This can be done in $O(n+m)$ time by traversing ST^* bottom-up. Note that $h_S\big(\ell(v)\big) = 1$ whenever the label of the arc $\big(\texttt{parent}(v), v\big)$ has length greater than one, and $h_S\big(\ell(v)\big) = f_S\big(\ell(v)\big)/f_S\big(\ell(\texttt{parent}(v))\big)$ otherwise. We can thus store in every node v the two integers $\texttt{hs} = h_S(\ell(v))$ and $\texttt{ht} = h_T(\ell(v))$, using an $O(n+m)$-time top-down traversal of ST^*. At this point, we can compute the contribution of all internal nodes of ST^* in overall $O(n+m)$ time, by accessing $v.\texttt{hs}$, $v.\texttt{ht}$, $\texttt{suffixLink}(v).\texttt{hs}$ and $\texttt{suffixLink}(v).\texttt{ht}$ for every v, and by computing $g(|S|, |\ell(v)|)$ and $g(|T|, |\ell(v)|)$ in constant time. □

In the second phase, we consider all substrings W that occur in S, in T, or in both, and whose label corresponds to a path in ST^* that *ends in the middle of an arc*. It is particularly easy to process strings that can be obtained by extending to the right by one character the label of an internal node of ST^*.

LEMMA 11.16 *Given* ST^**, we can compute in* $O(n+m)$ *time the values of* $z_S(W)$ *and of* $z_T(W)$ *for all strings* W *such that*

- W *occurs in* S *or in* T*;*
- $W = \ell(u) \cdot c$ *for some* $c \in [1..\sigma]$ *and for some internal node* u *of* ST^**;*
- $\ell(v) \neq W$ *for all nodes* v *of* ST^*.

Proof Assume that $W = \ell(u) \cdot c$ ends in the middle of edge (u, v), and that the nodes of ST^* store the values described in Lemma 11.15. Then, $h_S(W) = v.\texttt{fs}/u.\texttt{fs}$ and $h_S(W[2..k]) = \texttt{child}(\texttt{suffixLink}(u), c).\texttt{fs}/\texttt{suffixLink}(u).\texttt{fs}$. The same holds for computing $z_T(W)$. □

Note that all the minimal absent words of S that do occur in T, and symmetrically all the minimal absent words of T that do occur in S, are handled by Lemma 11.16. All other strings W that end in the middle of an edge (u, v) of ST^* have the following key property.

Score of nonmaximal strings. Consider an edge (u, v) of ST^* with $|\ell(u, v)| > 1$, and let W be a string of length $k > |\ell(u)| + 1$ that ends in the middle of (u, v). Then, $z_S(W)$ is either zero or $g(n, k) - 1$.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

Proof If W does not occur in S, then neither does its prefix $W[1..k-1]$, thus W is not a minimal absent word of S and $z_S(W) = 0$. If W does occur in S, then $h_S(W) = 1$. Moreover, since $(\texttt{suffixLink}(u), \texttt{suffixLink}(v))$ is an edge of ST^* and $W[2..k]$ lies inside such an edge, we are guaranteed that $h_S(W[2..k]) = 1$ as well. □

This proof works because, in ST^*, the destinations of the implicit Weiner links of ST have been assigned corresponding explicit nodes: in the original suffix tree ST, $f_S(W[2..k])$ is not guaranteed to equal $f_S(W[2..k-1])$, because $\texttt{suffixLink}(u)$ might be connected to $\texttt{suffixLink}(v)$ by a *path*, rather than by a single edge: see Figure 11.4. The following result derives immediately from the score of nonmaximal strings.

LEMMA 11.17 *Given ST^*, we can compute in $O(n+m)$ time and in $O(n+m)$ space the contribution to Equation (11.1) of all strings W such that*

- *W occurs in S or in T;*
- *$W = \ell(u) \cdot X$ for some internal node u of ST^* and for some string X of length at least two;*
- *$\ell(v) \neq W$ for all nodes v of ST^*.*

Proof Consider again the variables N, D_S, and D_T, whose values at the end of the computation will be

$$N = \sum_{W \in [1..\sigma]^+} \mathbf{S}_z[W]\mathbf{T}_z[W],$$
$$D_S = \sum_{W \in [1..\sigma]^+} \mathbf{S}_z[W]^2,$$
$$D_T = \sum_{W \in [1..\sigma]^+} \mathbf{T}_z[W]^2.$$

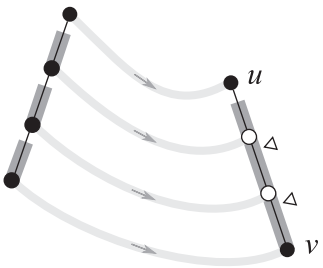

Figure 11.4 Illustrating the score of nonmaximal strings. Light gray arcs are Weiner links. Black dots are internal nodes in ST; white dots are destinations of implicit Weiner links. Both black and white dots are internal nodes in ST^*. Portions of edges of ST^* in which $h_S = 1$ are highlighted in dark gray. White triangles point to strings W for which $h_S(W)/h_S(W[2..|W|]) \neq 1$.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

By the score of nonmaximal strings, edge (u, v) gives the following contributions to N, D_S, and D_T:

$$N(u,v) = \sum_{i=|\ell(u)|+2}^{|\ell(v)|} \big(g(|S|,i)-1\big)\cdot\big(g(|T|,i)-1\big),$$

$$D_S(u,v) = \sum_{i=|\ell(u)|+2}^{|\ell(v)|} \big(g(|S|,i)-1\big)^2,$$

$$D_T(u,v) = \sum_{i=|\ell(u)|+2}^{|\ell(v)|} \big(g(|T|,i)-1\big)^2.$$

Assume that we have constant-time access to arrays $\texttt{cn}[1..n+m]$, $\texttt{cs}[1..n]$, and $\texttt{ct}[1..m]$ defined as follows:

$$\texttt{cn}[i] = \sum_{j=1}^{i} \big(g(n,j)-1\big)\cdot\big(g(m,j)-1\big),$$

$$\texttt{cs}[i] = \sum_{j=1}^{i} \big(g(n,j)-1\big)^2,$$

$$\texttt{ct}[i] = \sum_{j=1}^{i} \big(g(m,j)-1\big)^2.$$

Such arrays take $O(n+m)$ space and can be built in $O(n+m)$ time overall. Then, we can compute the contribution of any edge (u, v) in constant time, as follows:

$$N(u,v) = \texttt{cn}[|\ell(v)|] - \texttt{cn}[|\ell(u)|+1],$$
$$D_S(u,v) = \texttt{cs}[|\ell(v)|] - \texttt{cs}[|\ell(u)|+1],$$
$$D_T(u,v) = \texttt{ct}[|\ell(v)|] - \texttt{ct}[|\ell(u)|+1].$$

□

In the third phase of our algorithm, we consider the contribution of all the minimal absent words of S and of T *which occur neither in* S *nor in* T, or in other words that do not have a locus in ST^*. These are all the strings aWb which occur neither in S nor in T, but such that aW and Wb occur in S, in T, or in both. Note that the contribution of every such string to N, D_1, and D_2 is either zero or one. Proving the following lemma amounts to applying the strategy used to solve Exercise 11.4.

LEMMA 11.18 *We can compute in* $O((n+m)\sigma)$ *time the values of* $z_S(W)$ *and of* $z_T(W)$ *for all the minimal absent words* W *of* S *that do not occur in* T*, and for all the minimal absent words of* T *that do not occur in* S*.*

By combining Lemmas 11.15–11.18, we obtain the key result of this section.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

THEOREM 11.19 *Given two strings $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$, there is an algorithm that computes Equation (11.1) on the composition vectors $\mathbf{S}_z$ and $\mathbf{T}_z$ in $O((n+m)\sigma)$ time and $O(n+m)$ space.*

11.2.3 Substring kernels and matching statistics

The *matching statistics* of string $S \in [1..\sigma]^n$ with respect to string $T \in [1..\sigma]^m$ is a vector $\mathsf{MS}_{S,T}[1..n]$ that stores at position i the length of the longest prefix of $S[i..n]$ that occurs at least once in T. Functions over $\mathsf{MS}_{S,T}$ can be used to measure the similarity between S and T without resorting to alignment and without specifying a fixed string length k, and $\mathsf{MS}_{S,T}$ allows one to estimate the cross-entropy of the random processes that generated S and T: see Insight 11.5. Computationally, knowing $\mathsf{MS}_{S,T}$ allows one to derive a number of more specific string kernels, by indexing just one of S and T and by carrying out a linear scan of the other string.

Computing matching statistics

We can compute $\mathsf{MS}_{S,T}$ by scanning S from left to right while simultaneously issuing child and suffix-link queries on ST_T (see Figure 11.5(a)). Recall that this was called *descending suffix walk* in Exercise 8.22.

THEOREM 11.20 *Let $S \in [1..\sigma]^n$ and let $T \in [1..\sigma]^m$. Given ST_T, there is an algorithm that computes $\mathsf{MS}_{S,T}$ in $O(n \log \sigma)$ time using just the operations* `child` *and* `suffixLink` *on ST_T.*

Insight 11.5 Matching statistics and cross-entropy

Consider a random process with finite memory that generates sequences of characters on alphabet $\Sigma = [1..\sigma]$ according to a probability distribution $\mathbb{P}$. Specifically, let $X = X_1 X_2 \cdots X_n$ be a sequence of random characters on the alphabet Σ, and for a given string $S = S_1 S_2 \cdots S_n \in [1..\sigma]^n$, let $\mathbb{P}(S)$ be the probability that $X_i = S_i$ for $i \in [1..n]$. The entropy of the probability distribution $\mathbb{P}$ is

$$\mathsf{H}(\mathbb{P}) = \lim_{n\to\infty} \frac{-\sum_{S\in[1..\sigma]^n} \mathbb{P}(S)\log \mathbb{P}(S)}{n}.$$

Intuitively, it represents the average number of bits per character needed to identify a string generated by $\mathbb{P}$. Consider now a different random process that generates sequences of characters on the alphabet Σ with a probability distribution $\mathbb{Q}$. The *cross-entropy* of $\mathbb{P}$ with respect to $\mathbb{Q}$ is

$$\mathsf{H}(\mathbb{P},\mathbb{Q}) = \lim_{n\to\infty} \frac{-\sum_{S\in[1..\sigma]^n} \mathbb{P}(S)\log \mathbb{Q}(S)}{n}.$$

Intuitively, it measures the average number of bits per character that are needed to identify a string generated by $\mathbb{P}$, using a coding scheme based on $\mathbb{Q}$. Note that $\mathsf{H}(\mathbb{P},\mathbb{P}) = \mathsf{H}(\mathbb{P})$. A related notion is the Kullback–Leibler divergence of two probability distributions (see Section 6.4):

$$\mathsf{KL}(\mathbb{P},\mathbb{Q}) = \lim_{n\to\infty} \frac{\sum_{S\in[1..\sigma]^n} \mathbb{P}(S)\cdot(\log \mathbb{P}(S) - \log \mathbb{Q}(S))}{n}.$$

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

Intuitively, it measures the expected number of *extra bits* per character required to identify a string generated by $\mathbb{P}$, using a coding scheme based on $\mathbb{Q}$. Note that $\mathsf{KL}(\mathbb{P},\mathbb{Q}) = \mathsf{H}(\mathbb{P},\mathbb{Q}) - \mathsf{H}(\mathbb{P})$. Since $\mathsf{KL}(\mathbb{P},\mathbb{Q}) \neq \mathsf{KL}(\mathbb{Q},\mathbb{P})$ in general, $d_{KL}(\mathbb{P},\mathbb{Q}) = \mathsf{KL}(\mathbb{P},\mathbb{Q}) + \mathsf{KL}(\mathbb{Q},\mathbb{P})$ is typically preferred to quantify the dissimilarity between two probability distributions: indeed, $d_{KL}(\mathbb{P},\mathbb{Q})$ is symmetric, non-negative, and it equals zero when $\mathbb{P} = \mathbb{Q}$.

Assume now that we have a finite sequence $Y = Y_1 \cdots Y_m$ of m observations of the random process with probability distribution $\mathbb{Q}$, and assume that we are receiving a sequence of observations $X = X_1 X_2 \cdots$ from the random process with probability distribution $\mathbb{P}$. Let ℓ be the length of the *longest prefix* of $X_1 X_2 \cdots$ which occurs at least once in Y, and let $\mathbb{E}_{\mathbb{P},\mathbb{Q}}(\ell)$ be the expected value of ℓ under distributions $\mathbb{P}$ and $\mathbb{Q}$. It can be shown that

$$\lim_{m\to\infty} \left| \mathbb{E}_{\mathbb{P},\mathbb{Q}}(\ell) - \frac{\log m}{\mathsf{H}(\mathbb{P},\mathbb{Q})} \right| \in O(1).$$

Therefore, we can estimate $\mathsf{H}(\mathbb{P},\mathbb{Q})$ by $\log m / \mathbb{E}_{\mathbb{P},\mathbb{Q}}(\ell)$, and $\mathsf{H}(\mathbb{Q})$ by $\log m / \mathbb{E}_{\mathbb{P}}(\ell)$. If the sequence $X = X_1 X_2 \cdots X_m$ is finite but long enough, we can estimate ℓ by $\tilde{\ell} = \sum_{i=1}^{m} \mathsf{MS}_{X,Y}[i]/m$, where $\mathsf{MS}_{X,Y}$ is the vector of the matching statistics of X with respect to Y (clearly $\tilde{\ell} = (n+1)/2$ if X and Y coincide). These estimators for $\mathsf{H}(\mathbb{P},\mathbb{Q})$, $\mathsf{KL}(\mathbb{P},\mathbb{Q})$, and ultimately for $d_{KL}(\mathbb{P},\mathbb{Q})$, are known as *average common substring* estimators. The estimator for $d_{KL}(\mathbb{P},\mathbb{Q})$ has been used to build accurate phylogenetic trees from the genomes and proteomes of hundreds of evolutionarily distant species and thousands of viruses.

Proof Assume that we are at position i in S, and let $W = S[i..i + \mathsf{MS}_{S,T}[i] - 1]$. Note that W can end in the middle of an edge (u, v) of ST: let $W = aXY$, where $a \in [1..\sigma]$, $X \in [1..\sigma]^*$, $aX = \ell(u)$, and $Y \in [1..\sigma]^*$. Moreover, let $u' = \texttt{suffixLink}(u)$ and $v' = \texttt{suffixLink}(v)$. Note that suffix links can project arc (u, v) onto a *path* $u', v_1, v_2, \ldots, v_k, v'$. Since $\mathsf{MS}_{S,T}[i+1] \geq \mathsf{MS}_{S,T}[i] - 1$, the first step to compute $\mathsf{MS}_{S,T}[i+1]$ is to find the position of XY in ST: we call this phase of the algorithm the *repositioning phase*. To implement the repositioning phase, it suffices to take the suffix link from u, follow the outgoing arc from u' whose label starts with the first character of Y, and then iteratively jump to the next internal node of ST, choosing the next edge according to the corresponding character of Y. After the repositioning phase, we start matching the new characters of S on ST, or equivalently we read characters $S[i + \mathsf{MS}_{S,T}[i]], S[i + \mathsf{MS}_{S,T}[i] + 1], \ldots$ until the corresponding right-extension is impossible in ST. We call this phase of the algorithm the *matching phase*. Note that no character of S that has been read during the repositioning phase of $\mathsf{MS}_{S,T}[i+1]$ will be read again during the repositioning phase of $\mathsf{MS}_{S,T}[i+k]$ with $k > 1$: it follows that every position j of S is read at most twice, once in the matching phase of some $\mathsf{MS}_{S,T}[i]$ with $i \leq j$, and once in the repositioning phase of some $\mathsf{MS}_{S,T}[k]$ with $i < k < j$. Since every mismatch can be charged to the position of which it concludes the matching statistics, the total number of mismatches encountered by the algorithm is bounded by the length of S. □

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

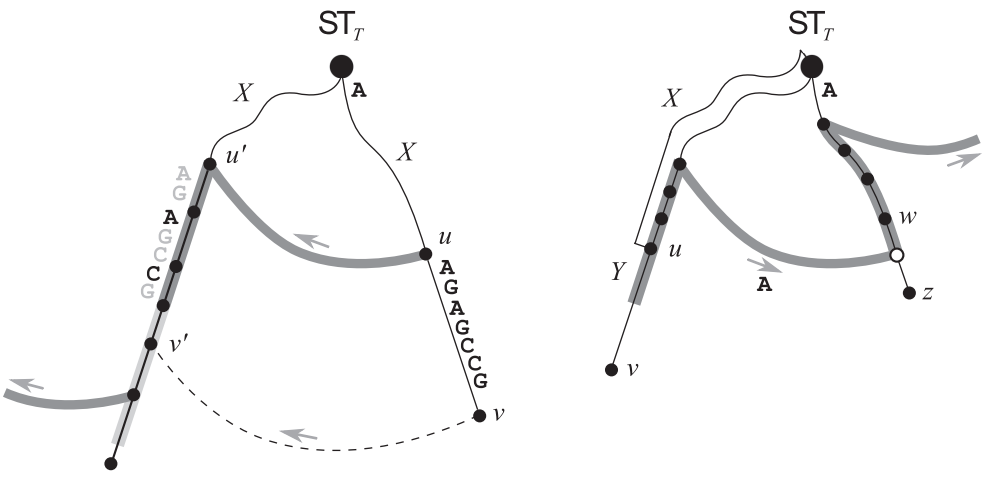

Figure 11.5 Two ways of computing $\mathsf{MS}_{S,T}$ on the suffix tree of T. (a) Using suffix links and child operations (Theorem 11.20). The picture shows substring $S[i..i + \mathsf{MS}_{S,T}[i] - 1] = \mathtt{A}XY$, where $Y = \mathtt{AGAGCCG}$. Dark gray lines mark the repositioning phase; light gray lines mark the matching phase. The characters of Y that are read during the repositioning phase are highlighted in black on edge (u', v'). Arrows indicate the direction of suffix links. (b) Using Weiner links and parent operations (Theorem 11.21). The picture shows substring $S[i..i + \mathsf{MS}_{S,T}[i] - 1] = XY$, where $S[i - 1] = \mathtt{A}$. Dark gray lines mark the traversal of Theorem 11.21. Arrows indicate the direction of Weiner links.

We can also compute $\mathsf{MS}_{S,T}$ symmetrically, by scanning S from right to left and by following parent links and Weiner links on ST_T (see Figure 11.5(b)).

THEOREM 11.21 *Let $S \in [1..\sigma]^n$ and let $T \in [1..\sigma]^m$. Given ST_T, there is an algorithm that computes $\mathsf{MS}_{S,T}$ in $O(n \log \sigma)$ time using just the operations* `parent` *and* `weinerLink` *on ST_T.*

Proof We work with the extended version of ST_T described in Section 8.3, in which every destination of an implicit Weiner link is assigned to a corresponding artificial node. Every such node has exactly one child, and we further assume that it stores a pointer to its highest descendant with at least two children. We also assume that the operation `parent`(v) returns the lowest ancestor of node v with at least two children.

Assume that we are at position i in S, let u and v be defined as in Theorem 11.20, and let $S[i..i + \mathsf{MS}_{S,T}[i] - 1] = XY$, where $X = \ell(u)$ and $Y \in [1..\sigma]^*$. To compute $\mathsf{MS}_{S,T}[i - 1]$ we can proceed as follows. We read character $a = S[i - 1]$, and we check the existence of an explicit or implicit Weiner link `weinerLink`$(v, a) = v'$ labeled by a from node v. If such a link exists, we set $\mathsf{MS}_{S,T}[i - 1] = \mathsf{MS}_{S,T}[i] + 1$. If the link is implicit, or equivalently if v' is a unary node that lies inside some edge (w, z) of the original suffix tree ST, then aXY lies inside edge (w, z) of ST as well, and we can find z in constant time by following the pointer stored in v'. If the link is explicit, or equivalently if v' has at least two children, then aXY lies inside the

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

edge $(\texttt{parent}(v'), v')$. If no Weiner link labeled by a exists from v, then aXY does not occur in T. Thus we iteratively reset v to $\texttt{parent}(v)$ and we try a Weiner link $\texttt{weinerLink}(v, a) = v'$ labeled by a from v: if such a link exists, we set $\mathsf{MS}_{S,T}[i-1] = |\ell(v)|+1$. Clearly every move from v to $\texttt{parent}(v)$ can be charged to a distinct position in S, thus the algorithm performs at most n calls to `parent` in total. □

As a byproduct, these approaches yield the nodes $v_i = \texttt{locus}(W_i)$ and $u_i = \text{argmax}\{|\ell(u')| : W_i = \ell(u') \cdot X, X \in [1..\sigma]^*\}$ for every $W_i = S[i..i + \mathsf{MS}_{S,T}[i] - 1]$ and $i \in [1..n]$. In what follows, we assume that pointers to such nodes of ST_T are stored in the arrays $\texttt{floor}_{S,T}[1..n]$ and $\texttt{ceil}_{S,T}[1..n]$, or equivalently that $\texttt{floor}_{S,T}[i] = u_i$ and $\texttt{ceil}_{S,T}[i] = v_i$ for every $i \in [1..n]$.

These algorithms on ST_T can also be adapted to compute the *shortest unique substring array* $\mathsf{SUS}_T[1..m]$, where $\mathsf{SUS}_T[i]$ contains the length of the shortest substring that starts at position i in T and that occurs exactly once in T, or equivalently the length of the shortest string that identifies position i uniquely among all positions in T: see Exercise 11.20. Shortest unique substrings have applications in a variety of sequence identification tasks, for example in the design of primer sequences for targeted resequencing (see Section 1.3), in the development of antibodies, and in comparative genomics.

Deriving substring kernels from matching statistics

Assume that $\alpha : [1..\sigma]^+ \mapsto \mathbb{R}$ is a function that assigns a weight to every string on alphabet $[1..\sigma]$. Given two strings $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$, let $\beta : [1..n] \mapsto \mathbb{R}$ and $\gamma : [1..m] \mapsto \mathbb{R}$ be functions that assign a weight to every position of S and T, respectively. Assume that we want to compute the following generalization of the substring kernel between S and T described in Definition 11.9:

$$\kappa(S, T, \alpha, \beta, \gamma) = \sum_{W \in [1..\sigma]^+} \alpha(W) \cdot \left(\sum_{i \in \mathcal{L}(W,S)} \beta(i) \right) \cdot \left(\sum_{i \in \mathcal{L}(W,T)} \gamma(i) \right), \qquad (11.4)$$

where $\mathcal{L}(W, X)$ is the set of all starting positions of occurrences of string W in string X. Moreover, if W is a substring of T that ends inside edge (u, v) of ST_T, let

$$f(W, T) = \sum_{j=|\ell(u)|+1}^{|W|} \alpha(W[1..j]).$$

Computing Equation (11.4) is particularly easy using matching statistics.

THEOREM 11.22 *Assume that we are given ST_T, and that we can compute $f(W, T)$ in constant time for every substring W of T. Then, Equation (11.4) can be computed in $O(n \log \sigma)$ time.*

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

Proof Assume that a substring W of T occurs at position i in S. Observing W at position i in S contributes quantity $\beta(i) \cdot \texttt{contrib}(W)$ to $\kappa(S,T)$, where

$$\begin{aligned}\texttt{contrib}(W) &= \sum_{j=1}^{|W|} \alpha\big(W[1..j]\big) \cdot \left(\sum_{k \in \mathcal{L}(W[1..j],T)} \gamma(k) \right) \\ &= \texttt{contrib}\big(\ell(u)\big) + \left(\sum_{k \in \mathcal{L}(\ell(v),T)} \gamma(k) \right) \cdot f(W,T). \end{aligned} \tag{11.5}$$

Therefore we can compute $\kappa(S,T)$ as follows:

$$\kappa(S,T,\alpha,\beta,\gamma) = \sum_{i=1}^{n} \beta(i) \cdot \texttt{contrib}\left(S[i..i + \mathsf{MS}_{S,T}[i] - 1]\right). \tag{11.6}$$

Recall that $\sum_{k \in \mathcal{L}(\ell(v),T)} \gamma(k)$ can be precomputed for every node v of ST_T in overall $O(m)$ time by a depth-first traversal of ST_T. Since we can compute $f(\ell(u),T)$ in constant time for every node u of ST, we can precompute $\texttt{contrib}\big(\ell(u)\big)$ and store it inside every node u of ST in overall $O(m)$ time, by a top-down traversal of ST that uses Equation (11.5). After this preprocessing, computing Equation (11.6) takes just $O(n \log \sigma)$ time using the algorithms described in Theorems 11.20 and 11.21. □

If $\alpha(W)$ depends just on the length of W rather than on its characters, and if $\alpha(W) = 0$ for $|W| > \tau$ for some threshold τ, we can access $f(W,T)$ in constant time after a $O(\tau)$-time and $O(\tau \log \tau)$-space precomputation: see Exercise 11.21. Precomputing is not even necessary in the following length-based weighting schemes, since $f(W,T)$ has a closed form that can be evaluated in constant time (see Exercise 11.21):

- exponential decay, $\alpha(W) = \lambda^{-|W|}$ for some constant λ;
- constant weight, $\alpha(W) = \lambda|W|$;
- bounded range, $\alpha(W) = \lambda$ if $|W| \leq \tau$, and $\alpha(W) = 0$ otherwise;
- τ-spectrum, $\alpha(W) = \lambda$ if $|W| = \tau$, and $\alpha(W) = 0$ otherwise.

Theorem 11.22 can also be adapted to compute $\kappa(S,T,\alpha,\beta,\gamma)$ only on a user-specified subset of $[1..\sigma]^+$, with custom weights: see Exercise 11.22.

In most applications $\beta(i) = 1$ for every i, but varying β allows one to model some common kernels. For example, let $\{S^1, S^2, \ldots, S^k\}$ be a fixed set of strings with weights $\{w_1, w_2, \ldots, w_k\}$. Assume that we are given a query string T, and that we want to compute $\sum_{i=1}^{k} w_i \cdot \kappa(S^i,T,\alpha,\beta,\gamma)$, where $\beta(i) = 1$ for every i. This clearly coincides with $\kappa(S^1 \cdot S^2 \cdot \cdots \cdot S^k,T,\alpha,\beta',\gamma)$, where $\beta'(i) = w_j$ for all positions i between $\sum_{h=1}^{j-1} |S^h| + 1$ and $\sum_{h=1}^{j} |S^h|$.

* Matching statistics and variable-memory Markov chains

Consider a Markov chain of order k (see Chapter 7). Given a string $W \in [1..\sigma]^h$ with $h < k$, it could happen in practice that $\mathbb{P}(a|UW) = \mathbb{P}(a|VW)$ for all strings $U \neq V$ in $[1..\sigma]^{k-h}$ and for all $a \in [1..\sigma]$. In other words, adding up to $k-h$ characters *before* W does not alter the probability of seeing any character *after* W. In such cases, the Markov

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

chain is said to have *contexts of variable length*, and it can be represented in compact form by storing just a set of tuples $\mathcal{W} = \{(W, p_1, p_2, \ldots, p_\sigma)\}$, where W is a context and p_i is the probability of seeing character $i \in [1..\sigma]$ when W is the *longest suffix* of the generated string that equals the first component of a tuple in $\mathcal{W}$. We call such p_i *emission probabilities* of context W. In practice, set $\mathcal{W}$ is encoded as a trie $\mathsf{PST}_{\mathcal{W}}$, called a *probabilistic suffix trie*, in which contexts W are inserted *from right to left*, and in which every node of $\mathsf{PST}_{\mathcal{W}}$ that corresponds to a context W stores its emission probabilities.

Given a string $S \in [1..\sigma]^+$, the probability that S was generated by the variable-length Markov chain encoded by $\mathsf{PST}_{\mathcal{W}}$ is computed character by character, as follows: at every position j of S, the probability that $\mathsf{PST}_{\mathcal{W}}$ generated character $S[j]$ is determined by finding the *longest substring* $S[i..j-1]$ that labels a node v of $\mathsf{PST}_{\mathcal{W}}$ that corresponds to a context of $\mathcal{W}$, and by reading the emission probability of character $S[j]$ stored at v. Such a longest substring $S[i..j-1]$ is found by reading the characters $S[j-1], S[j-2], \ldots$ starting from the root of $\mathsf{PST}_{\mathcal{W}}$. The probability of S given $\mathsf{PST}_{\mathcal{W}}$ is the product of the probabilities of all of the characters of S. It can be shown that the probability distribution on $[1..\sigma]^+$ induced by a probabilistic suffix trie is equivalent to the probability distribution on $[1..\sigma]^+$ induced by the variable-length Markov chain of order k of which the trie is a compact representation. In what follows, we drop the subscript from $\mathsf{PST}_{\mathcal{W}}$ whenever $\mathcal{W}$ is implicit from the context.

Assume that we have a string or a concatenation of related strings T, for example genomes of similar species or proteins with the same function or structure. A PST for T can be interpreted as a *generative model* of T: given a new string S, measuring the probability that S was generated by PST is a way to measure the similarity between the random process that generated S and the random process that generated T, without resorting to alignment. This is conceptually analogous to Problem 7.4 in hidden Markov models.

In practice, it is natural to build PST just for the set of contexts defined as follows.

DEFINITION 11.23 *Let aW be a substring of a string $T \in [1..\sigma]^n$, where $a \in [1..\sigma]$ and $W \in [1..\sigma]^*$. We say that aW is a* context *if and only if all the following conditions hold.*

i. *Given a user-specified threshold $\tau_1 > 0$,*

$$\frac{f_T(aW)}{|T| - |aW| + 1} \geq \tau_1.$$

ii. *There is a character $b \in [1..\sigma]$ such that*
(a) given a user-specified threshold $\tau_2 \leq 1$,

$$\frac{f_T(aWb)}{f_T(aW)} \geq \tau_2;$$

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

(b) *given user-specified thresholds* $\tau_3 > 1$ *and* $\tau_4 < 1$, *either* $\alpha \geq \tau_3$ *or* $\alpha \leq \tau_4$, *where*

$$\alpha = \frac{f_T(aWb)/f_T(aW)}{f_T(Wb)/f_T(W)}.$$

Surprisingly, the set of strings that satisfy condition ii is a subset of the nodes of the suffix tree of T.

LEMMA 11.24 *Given a string* $T \in [1..\sigma]^n$ *and the suffix tree* ST_T, *we can determine all context in Definition 11.23, and compute their emission probabilities, in overall* $O(n)$ *time.*

Proof If string aW ends in the middle of an edge of ST_T, the left-hand side of Equation (11.7) equals one, therefore condition ii(a) is satisfied. However, if even W ends in the middle of an edge, that is, if aW is not the destination of an implicit Weiner link, then $\alpha = 1$ and condition ii(b) is not satisfied. It follows that, given the augmented version ST_T^* of ST_T in which all destinations of implicit Weiner links have been assigned a unary node, we can mark all nodes that correspond to contexts, and annotate them with emission probabilities, in overall $O(n)$ time. □

The following equivalence between the PST of the contexts of T and the suffix-link tree of T is thus evident.

COROLLARY 11.25 *Let* SLT_T^* *be the* suffix-link tree *of* T, *extended with implicit Weiner links and their destinations. The* PST *of the set of contexts in Definition 11.23 is the subtree of* SLT_T^* *defined by the following procedure: starting from every node of* ST_T^* *that corresponds to a context, recursively follow suffix links up to the root of* ST_T^*, *marking all nodes met in the process.*

Corollary 11.25 enables a simple, linear-time algorithm for computing the probability of a query string S, based on matching statistics.

THEOREM 11.26 *Given the suffix tree* $\mathsf{ST}_{\underline{T}}$ *of the reverse of* $T \in [1..\sigma]^n$, *and the suffix-link tree* SLT_T^* *of* T *with nodes marked as in Lemma 11.24, we can compute the probability of a string* $S \in [1..\sigma]^m$ *with respect to the* PST *of* T *in* $O(n + m \log \sigma)$ *time.*

Proof We add to $\mathsf{ST}_{\underline{T}}$ a node for every label W of a node in SLT_T^*, if string W does not already label a node in $\mathsf{ST}_{\underline{T}}$. Then, we add to every node in such an augmented suffix tree a pointer to its closest ancestor labeled by $\underline{W}$, where W is a context of T. Such an ancestor might be the node itself, or it might not exist. The size of the augmented suffix tree is still $O(n)$, and it can be built in $O(n)$ time: see Exercise 11.24. Then, we compute the matching statistics of $\underline{S}$ with respect to $\underline{T}$, obtaining for every position j in S a pointer to the locus $\texttt{ceil}_{\underline{S},\underline{T}}[j]$ in $\mathsf{ST}_{\underline{T}}$ of the longest substring of T that ends at position j in S. From this node we can reach in constant time the node associated with the longest context of T that labels a suffix of $S[1..j]$, thus deriving the probability of character $S[j+1]$. □

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

11.2.4 Mismatch kernels

Consider a string $S \in [1..\sigma]^+$, let $\Gamma(S,k)$ be the set of all distinct k-mers that occur exactly in S, and let $f_S(W,m)$ be the number of (possibly overlapping) occurrences of a k-mer W in S *with at most m mismatches*. Taking into account mismatches is of key importance for analyzing genomes, where sequential signals (like the transcription factor binding sites described in Section 1.1) are often inexact due to genetic variations. Given two strings S and T on alphabet $[1..\sigma]$, we want to compute the following generalization of the k-mer kernel described in Section 11.2.1:

$$\kappa(S,T,k,m) = \sum_{W \in [1..\sigma]^k} f_S(W,m) \cdot f_T(W,m). \tag{11.7}$$

Given two k-mers V and W on an alphabet $[1..\sigma]$, consider the Hamming distance $D_{\mathsf{H}}(V,W)$ between V and W defined in Section 6.1, and let $B(W,m)$ be the ball of $[1..\sigma]^k$ with center W and radius m, or in other words the subset of $[1..\sigma]^k$ that lies at Hamming distance at most m from W. Note that the size of $B(V,m) \cap B(W,m)$ depends only on $D_{\mathsf{H}}(V,W)$, so let $g(k,d,m)$ be the mapping from the Hamming distance d of two k-mers to the size of the intersection of their Hamming balls of radius m, where $g(k,d,m) = 0$ whenever $d > \min\{2m,k\}$. In this section we assume that we have a table $\texttt{kmersToDistance}_k[1..\sigma^{2k}]$ that contains $D_{\mathsf{H}}(V,W)$ for every pair of k-mers V and W on alphabet $[1..\sigma]$, and a table $\texttt{distanceToSize}_{k,m}[1..\min\{2m,k\}]$ that contains $g(k,d,m)$ for every d that occurs in $\texttt{kmers2distance}_k$. Note that, for a specific choice of k and d, the value $g(k,d,m)$ has a closed-form representation that can be computed in $O(m)$ time: see Exercise 11.25.

To warm up, we consider an $O(|\Gamma(S,k)| \cdot |\Gamma(T,k)|)$-time algorithm to compute Equation (11.7).

LEMMA 11.27 *Given the sets $\Gamma(S,k)$ and $\Gamma(T,k)$ of two strings S and T on an alphabet $[1..\sigma]$, augmented with the frequency of their distinct k-mers, and given tables $\texttt{kmersToDistance}_k$ and $\texttt{distanceToSize}_{k,m}$, there is an algorithm that computes Equation (11.7) in $O(|\Gamma(S,k)| \cdot |\Gamma(T,k)|)$ time.*

Proof After simple manipulations, Equation (11.7) becomes

$$\begin{aligned}\kappa(S,T,k,m) &= \sum_{V \in \Gamma(S,k)} \sum_{W \in \Gamma(T,k)} f_S(V) \cdot f_T(W) \cdot |B(V,m) \cap B(W,m)| \\ &= \sum_{V \in \Gamma(S,k)} \sum_{W \in \Gamma(T,k)} f_S(V) \cdot f_T(W) \cdot g(k, D_{\mathsf{H}}(V,W), m), \end{aligned} \tag{11.8}$$

where $f_S(W)$ is the number of *exact* occurrences of W in S (see the beginning of Section 11.2). We can thus compute $D_{\mathsf{H}}(V,W)$ in constant time for every pair of k-mers V and W using the table $\texttt{kmersToDistance}_k$, and we can compute $g(k, D_{\mathsf{H}}(V,W), m)$ in constant time using the table $\texttt{distanceToSize}_{k,m}$. □

The time complexity of Lemma 11.27 can be improved by reorganizing the computation of Equation (11.8).

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

THEOREM 11.28 *Given the sets $\Gamma(S,k)$ and $\Gamma(T,k)$ of two strings S and T on alphabet $[1..\sigma]$, augmented with the frequency of their distinct k-mers, and given table* `distanceToSize`$_{k,m}$*, there is an algorithm that computes Equation (11.7) in $O(\sigma + |\Gamma(S,k)| + |\Gamma(T,k)|)$ time.*

Proof Recall that $g(k,d,m) = 0$ for $d > \min\{2m,k\}$. We can thus rewrite Equation (11.8) as

$$\kappa(S,T,k,m) = \sum_{d=0}^{\min\{2m,k\}} g(k,d,m) \cdot P_d,$$
$$P_d = \sum_{\substack{V\in\Gamma(S,k)\\ W\in\Gamma(T,k)\\ D_{\mathsf{H}}(V,W)=d}} f_S(V) \cdot f_T(W). \tag{11.9}$$

In other words, computing Equation (11.8) reduces to computing P_d for all $d \in [0..\min\{2m,k\}]$.

Assume that we remove a specific subset of d positions from all k-mers in $\Gamma(S,k)$ and $\Gamma(T,k)$. As a result, some distinct k-mers in $\Gamma(S,k)$ and in $\Gamma(T,k)$ can collapse into identical $(k-d)$-mers. We compute the inner product C_d between such lists of $|\Gamma(S,k)|$ and $|\Gamma(T,k)|$, potentially repeated $(k-d)$-mers, in overall $O\big(\sigma + (k-d)\cdot(|\Gamma(S,k)| + |\Gamma(T,k)|)\big)$ time, by sorting the $(k-d)$-mers in each list into lexicographic order using Lemma 8.7, and by merging the sorted lists. Then, we repeat this process for all the possible selections of d positions out of k, adding all the $\binom{k}{d}$ possible inner products to C_d.

Note that, if $D_{\mathsf{H}}(V,W) = i < d$ for some k-mers V and W, $f_S(V)f_T(W)$ is added to C_d when processing $\binom{k-i}{d-i}$ subsets of d positions. It follows that $P_d = C_d - \sum_{i=0}^{d-1}\binom{k-i}{d-i}P_i$, therefore we can compute Equation (11.9) in overall $O(a_{k,m}\cdot\sigma + b_{k,m}\cdot(|\Gamma(S,k)| + |\Gamma(T,k)|) + c_{k,m})$ time, where

$$a_{k,m} = \sum_{d=0}^{\min\{2m,k\}} \binom{k}{d},$$
$$b_{k,m} = \sum_{d=0}^{\min\{2m,k\}} \binom{k}{d}(k-d),$$
$$c_{k,m} = \sum_{d=0}^{\min\{2m,k\}} \sum_{i=0}^{d-1} \binom{k-i}{d-i} + \min\{2m,k\}.$$

□

Note that when $d = k$ we can avoid sorting and merging altogether, since $P_k = (|S|-k+1)\cdot(|T|-k+1) - \sum_{i=0}^{k-1} P_i$. The algorithm in Theorem 11.28 can be adapted to compute other popular inexact kernels for biosequence comparison, such as the *gapped kernel* and the *neighborhood kernel*: see Exercises 11.26 and 11.27.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

11.2.5 Compression distance

Another way of quantifying the similarity of two strings S and T without resorting to alignment consists in using a *compressor*: intuitively, if S and T are very similar to each other, compressing them together should produce a smaller file than compressing each of them in isolation. This intuition can be formalized in the notion of *Kolmogorov complexity*, an algorithmic measure of information that defines the complexity of a string S as the length $K(S)$ of *the shortest program that outputs* S in a reference computer. The theory of Kolmogorov complexity is vast and theoretically deep: in what follows we avoid the abstraction of shortest-possible programs, and instead focus on real compressors. Measures of complexity and similarity based on real compressors have been used to detect low-complexity regions in genomes, and to build phylogenies of mitochondrial and viral genomes. Like string kernels, a compression-based measure of similarity allows one also to compare heterogeneous objects, like sequenced genomes and read sets.

Consider a real, *lossless* compressor C, that is, a compressor that allows one to reconstruct its input from its output *exactly*, like the Lempel–Ziv compressor described in Chapter 12. Let $C(S)$ be the number of bits in the output when the input of the compressor is string $S \in [1..\sigma]^n$, and assume that $C(S) \leq |S| + O(\log|S|)$. Given another string $T \in [1..\sigma]^m$, let $C(T|S) = C(ST) - C(S)$ be the number of bits that must be added to the compressed representation of S to encode the compressed representation of T. Similarly, let $C(T) - C(T|S)$ be the number of bits in the compressed representation of T that cannot be derived from the compressed representation of S.

In what follows we focus on *normal compressors*.

DEFINITION 11.29 *A compressor C is* normal *if it satisfies the following properties for all strings S, T, and U, up to an additive $O(\log n)$ term, where n is the size of the input in bits:*

- idempotency, $C(SS) = C(S)$;
- monotonicity, $C(ST) \geq C(S)$;
- symmetry, $C(ST) = C(TS)$;
- $C(S|T) \leq C(S|U) + C(U|T)$.

Symmetry might be violated by on-line compressors, like the Lempel–Ziv algorithm described in Section 12.1: while reading ST from left to right, for example, the Lempel–Ziv algorithm first adapts to the regularities of S; then, after switching to T, it adapts to the regularities of T only after a number of iterations. If S and T are sufficiently large, such violations of symmetry are negligible in practice. Compressors based on the Burrows–Wheeler transform tend to show even smaller departures from symmetry.

The last property of a normal compressor is intuitively desirable, and it immediately implies *distributivity* (see Exercise 11.28). In turn, distributivity implies *subadditivity*:

- *distributivity*, $C(ST) + C(U) \leq C(SU) + C(TU)$;
- *subadditivity*, $C(ST) \leq C(S) + C(T)$.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

These properties are again valid up to an additive $O(\log n)$ term, where n is the size of the input in bits. Distributivity is satisfied by a number of real compressors, and it will be used repeatedly in what follows. Subadditivity is intuitively desirable, since a compressor should be able to use information present in S to better compress T. Again, minor imperfections to subadditivity might arise in practice, but they typically vanish when S and T become large enough.

We define the dissimilarity between two strings S and T using a normal compressor.

DEFINITION 11.30 *Let C be a normal compressor. The* normalized compression distance *between two strings S and T is*

$$\mathsf{NCD}(S,T) = \frac{C(ST) - \min\{C(S), C(T)\}}{\max\{C(S), C(T)\}}.$$

Note that $\mathsf{NCD}(S,T) \in [0..1+\epsilon]$, where $\epsilon \in O(\log n)$ depends on specific features of the compressor. Note also that, if $C(S) < C(T)$, $\mathsf{NCD}(S,T) = \big(C(ST) - C(S)\big)/C(T)$ measures the improvement which accrues from compressing T together with S, with respect to compressing T in isolation. In the following theorem we show that NCD is a *metric*. Recall from Exercise 6.1 that a measure of dissimilarity D is a metric if it satisfies the following properties for all strings S, T, and U:

- *non-negativity*, $D(S,T) \geq 0$;
- *identity of indiscernibles*, $D(S,T) = 0$ if and only if $S = T$;
- *symmetry*, $D(S,T) = D(T,S)$;
- *triangle inequality*, $D(S,T) \leq D(S,U) + D(U,T)$.

THEOREM 11.31 NCD *is a metric up to an $O(\log n)$ additive term, where n is the number of bits in the input of C.*

Proof $\mathsf{NCD}(S,T) \geq 0$ by the monotonicity and symmetry of C. $\mathsf{NCD}(S,S) = 0$ by the idempotency of C. $\mathsf{NCD}(S,T) = \mathsf{NCD}(T,S)$ by the symmetry of C. We thus need to show only the triangle inequality $\mathsf{NCD}(S,T) \leq \mathsf{NCD}(S,U) + \mathsf{NCD}(U,T)$ for every set of strings $\{S,T,U\} \subset [1..\sigma]^+$.

Without loss of generality, assume that $C(S) \leq C(T) \leq C(U)$: since NCD is symmetric, we need to prove the triangle inequality just for $\mathsf{NCD}(S,T)$, $\mathsf{NCD}(S,U)$, and $\mathsf{NCD}(T,U)$. Here we prove just that $\mathsf{NCD}(S,T) \leq \mathsf{NCD}(S,U) + \mathsf{NCD}(U,T)$, leaving the other cases to Exercise 11.29. By distributivity, we have that

$$C(ST) + C(U) \leq C(SU) + C(TU)$$

and by symmetry

$$C(ST) + C(U) \leq C(SU) + C(UT).$$

Subtracting $C(S)$ from both sides of the inequality, we obtain

$$\begin{aligned} C(ST) + C(U) - C(S) &\leq C(SU) + C(UT) - C(S), \\ C(ST) - C(S) &\leq C(SU) - C(S) + C(UT) - C(U), \end{aligned}$$

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

and on dividing both sides by $C(T)$ we have

$$\frac{C(ST) - C(S)}{C(T)} \leq \frac{C(SU) - C(S) + C(UT) - C(U)}{C(T)}.$$

By subadditivity, the left-hand side of this last inequality is at most one. If the right-hand side is at most one, then adding any positive number δ to both its numerator and its denominator can only increase the ratio. If the right-hand side is greater than one, then adding δ to the numerator and the denominator decreases the ratio, but the ratio still remains greater than one. It follows that adding a positive number δ to both the numerator and the denominator of the right-hand side keeps it greater than or equal to the left-hand side. We thus add the number δ that satisfies $C(U) = C(T) + \delta$, obtaining

$$\frac{C(ST) - C(S)}{C(T)} \leq \frac{C(SU) - C(S)}{C(U)} + \frac{C(UT) - C(T)}{C(U)}.$$

□

11.3 Literature

The algorithms presented in this chapter for maximal repeats, maximal unique matches, and maximal exact matches are mostly from Belazzougui *et al.* (2013); the solution for MUMs on multiple strings was inspired by the non-compact solution in Section 8.4.2 and the cross-product computation for MEMs is analogous to the one in Baker *et al.* (1993). Alternative ways to solve some of these problems using a BWT index were given earlier in Beller *et al.* (2012). Another space-efficient solution to MUMs is sketched in Exercise 11.3, following ideas from Hon & Sadakane (2002) and Fischer *et al.* (2008). Maximal unique matches and the heuristic alignment method described in Insight 11.1 are from Delcher *et al.* (1999). Minimal absent words have been extensively studied: see for example Crochemore *et al.* (1998) for additional formal properties, and see Crochemore *et al.* (2000), Hampikian & Andersen (2007), Herold *et al.* (2008), Chairungsee & Crochemore (2012), Garcia & Pinho (2011), Garcia *et al.* (2011), and references therein for a small set of applications to data compression and to the analysis and comparison of genomes.

We just scratched the surface of the vast area of string kernels, without giving details about its large array of applications to molecular biology. The space-efficient computation of substring and k-mer kernels described in Section 11.2.1 was developed for this book: for more details, see Sobih *et al.* (2014). The suffix tree computation of the substring kernel with Markovian corrections is described in Apostolico & Denas (2008), and its estimate of k-mer probability was first introduced and tested in Qi *et al.* (2004). In addition to Equation (11.3), other formulas for computing $\tilde{p}_S(W)$ – the probability of observing k-mer W under a null model for string S – have been proposed: see Vinga & Almeida (2003), Song *et al.* (2014) and references therein for additional estimates and distance measures. The idea of converting an estimation formula into its maximum-entropy version described in Insight 11.4 is detailed in Chan *et al.* (2012).

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

An extensive description of the statistical properties of some alignment-free distance measures applied to strings and read sets can be found in Reinert *et al.* (2009), Wan *et al.* (2010), and Song *et al.* (2013), and a geometric treatment of a specific instance of alignment-free sequence comparison is described in Behnam *et al.* (2013). The data-driven approach for choosing k detailed in Insight 11.3 is taken from Sims *et al.* (2009). Statistical analyses related to the properties studied in Exercise 11.14 can be found in Reinert *et al.* (2009).

The connection between matching statistics and cross-entropy mentioned in Insight 11.5 is described in Farach *et al.* (1995) and references therein. The first genome classification algorithm based on this connection was introduced and tested in Ulitsky *et al.* (2006). The two symmetrical ways of computing matching statistics are described in Ohlebusch *et al.* (2010), and the method for deriving substring kernels from matching statistics that we presented in Section 11.2.3 is from Smola & Vishwanathan (2003). For clarity we presented the latter algorithm on suffix trees: an implementation on suffix arrays is described in Teo & Vishwanathan (2006). The connection between suffix-link trees, matching statistics, and variable-length Markov chains was established in Apostolico & Bejerano (2000). See Ziv (2008) and references therein for statistical properties of probabilistic suffix tries applied to compression. It is clear from Section 11.2 that substring and k-mer kernels have strong connections to maximal substrings: an overview of this connection appears in Apostolico (2010).

Kernels that take into account the frequency of occurrences with mismatches, gaps of bounded or unbounded length, rigid wildcards, and character substitutions according to a given probabilistic model, as well as other forms of gap or mismatch scores, are detailed in Leslie & Kuang (2003), Haussler (1999), Lodhi *et al.* (2002) and references therein. Corresponding statistical corrections have been defined, for example in Göke *et al.* (2012). The computation of such kernels follows a quadratic-time dynamic programming approach (Lodhi *et al.* 2002; Cancedda *et al.* 2003; Rousu *et al.* 2005), a traversal of ad-hoc tries (Leslie & Kuang 2003), or a combination of the two, with complexity that grows exponentially with the number of allowed gaps or mismatches. The algorithm to compute mismatch kernels described in Section 11.2.4 appears in Kuksa *et al.* (2008) and Kuksa & Pavlovic (2012), where the authors solve Exercise 11.25 and describe how to compute a closed-form expression for the size of the intersection of Hamming balls for every setting of m and $h(V, W)$. For an in-depth description of kernel methods in learning theory, see Shawe-Taylor & Cristianini (2004) and Cristianini & Shawe-Taylor (2000).

Finally, the notions of a normal compressor and of the normalized compression distance are detailed in Cilibrasi & Vitányi (2005). A more powerful *normalized information distance* based on Kolmogorov complexity is described in Li *et al.* (2004). For a detailed exposition of the foundations of Kolmogorov complexity and of its wide array of applications, see Li & Vitányi (2008).

Exercises

11.1 Give an algorithm to construct the indicator bitvector I of Algorithm 11.2 in $O((m+n)\log\sigma)$ time and $O((m+n)\log\sigma)$ bits of space.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

11.2 Modify the algorithm for maximal unique matches on two strings to use two bidirectional indexes instead of the indicator bitvector as in our solution for maximal exact matches.

11.3 Recall the algorithm to compute maximal unique matches for multiple strings through document counting in Section 8.4.3. It is possible to implement this algorithm using $O(n \log \sigma)$ bits of space and $O(n \log \sigma \log n)$ time, but this requires some advanced data structures not covered in this book. Assume you have a data structure for solving the *dynamic partial sums* problem, that is, to insert, delete, and query elements of a list of non-negative integers, where the queries ask to sum values up to position i and to find the largest i such that the sum of values up to i is at most a given threshold x. There is a data structure for solving the updates and queries in $O(\log n)$ time using $O(n \log \sigma)$ bits of space, where n is the sum of the stored values. Assume also that you have a succinct representation of LCP values. There is a representation that uses $2n$ bits, such that the extraction of LCP[i] takes the same time as the extraction of SA[i]. Now, observe that the suffix array and the LCP array, with some auxiliary data structures, are sufficient to simulate the algorithm without any need for explicit suffix trees. Especially, differences between consecutive entries of string and node depths can be stored with the dynamic partial sums data structure. Fill in the details of this space-efficient algorithm.

11.4 Prove Theorem 11.7 assuming that in the output a minimal absent word aWb is encoded as a triplet $(i, j, |W|)$, where i (respectively j) is the starting position of an occurrence of aW in S (respectively Wb in S).

11.5 Prove that only a maximal repeat of a string $S \in [1..\sigma]^+$ can be the infix W of a minimal absent word aWb of S, where a and b are characters in $[1..\sigma]$.

11.6 Show that the number of minimal absent words in a string of length n over an alphabet $[1..\sigma]$ is $O(n\sigma)$. *Hint*. Use the result of the previous exercise.

11.7 A σ-ary de Bruijn sequence of order k is a circular sequence of length σ^k that contains all the possible k-mers over an alphabet $\Sigma = [1..\sigma]$. It can be constructed by spelling all the labels in an Eulerian cycle (a cycle that goes through all the edges) of a de Bruijn graph with parameters Σ and k. The sequence can be easily transformed into a string (non-circular sequence) of length $\sigma^k + k - 1$ that contains all the possible k-mers over alphabet Σ.

(a) Describe the transformation from circular sequence to string.
(b) Show that the number of minimal absent words in this string is σ^{k+1}.

11.8 Assume that you have a set of local alignments between two genomes A and B, with an alignment score associated with each alignment. Model the input as a bipartite graph where overlapping alignments in A and in B form a vertex, respectively, to two sides of the graph. Alignments form weighted edges. Which problem in Chapter 5 suits the purpose of finding anchors for rearrangement algorithms?

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

11.9 Recall that the formula $p_S(W) = f_F(W)/(|S| - |W| + 1)$ used in Section 11.2 to estimate the empirical probability of substring W of S assumes that W can occur at every position of T. Describe a more accurate expression for $p_S(W)$ that takes into account the *shortest period* of W.

11.10 The *Jaccard distance* between two sets $\mathcal{S}$ and $\mathcal{T}$ is defined as $J(\mathcal{S}, \mathcal{T}) = |\mathcal{S} \cap \mathcal{T}|/|\mathcal{S} \cup \mathcal{T}|$. Adapt the algorithms in Section 11.2.1 to compute $J(\mathcal{S}, \mathcal{T})$, both in the case where $\mathcal{S}$ and $\mathcal{T}$ are the sets of all distinct k-mers that occur in S and in T, respectively, and in the case in which $\mathcal{S}$ and $\mathcal{T}$ are the sets of all distinct *substrings*, of any length, that occur in S and in T, respectively.

11.11 Adapt Lemma 11.11 to compute a variant of the k-mer kernel in which $\mathbf{S}_k[W] = p_S(W)$ and $\mathbf{T}_k[W] = p_T(W)$ for every $W \in [1..\sigma]^k$.

11.12 Adapt Corollary 11.13 to compute a variant of the substring kernel in which $\mathbf{S}_\infty[W] = p_S(W)$ and $\mathbf{T}_\infty[W] = p_T(W)$ for every $W \in [1..\sigma]^k$.

11.13 Given a string S on alphabet $[1..\sigma]$ and a substring W of S, let $\texttt{right}(W)$ be the set of characters that occur in S after W. More formally, $\texttt{right}(W) = \{a \in [1..\sigma] \mid f_S(Wa) > 0\}$. The *kth-order empirical entropy* of S is defined as follows:

$$\mathsf{H}(S,k) = \frac{1}{|S|} \sum_{W \in [1..\sigma]^k} \sum_{a \in \texttt{right}(W)} f_S(Wa) \log\left(\frac{f_S(W)}{f_S(Wa)}\right).$$

Intuitively, $\mathsf{H}(S,k)$ measures the amount of uncertainty in predicting the character that follows a context of length k, thus it is a lower bound for the size of a compressed version of S in which every character is assigned a codeword that depends on the preceding context of length k. Adapt the algorithms in Section 11.2.1 to compute $\mathsf{H}(S,k)$ for $k \in [k_1..k_2]$ using the bidirectional BWT index of S, and state the complexity of your solution.

11.14 Let S and T be two strings on alphabet $[1..\sigma]$, and let $\mathbf{S}$ and $\mathbf{T}$ be their composition vectors indexed by all possible k-mers. When the probability distribution of characters in $[1..\sigma]$ is highly nonuniform, the inner product between $\mathbf{S}$ and $\mathbf{T}$ (also known as the D_2 *statistic*) is known to be dominated by noise. To solve this problem, raw counts are typically corrected by *expected counts* that depend on $p(W) = \prod_{i=1}^{|W|} p(W[i])$, the probability of observing string W if characters at different positions in S are independent, identically distributed, and have empirical probability $p(a) = f_{ST}(a)/(|S| + |T|)$ for $a \in [1..\sigma]$. For example, letting $\tilde{\mathbf{S}}[W] = \mathbf{S}[W] - (|S| - k + 1)p(W)$, the following variants of D_2 have been proposed:

$$D_2^s = \sum_{W \in [1..\sigma]^k} \frac{\tilde{\mathbf{S}}[W]\tilde{\mathbf{T}}[W]}{\sqrt{\tilde{\mathbf{S}}[W]^2 + \tilde{\mathbf{T}}[W]^2}},$$

$$D_2^* = \sum_{W \in [1..\sigma]^k} \frac{\tilde{\mathbf{S}}[W]\tilde{\mathbf{T}}[W]}{\sqrt{(|S| - k + 1)(|T| - k + 1) \cdot p(W)}}.$$

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

Adapt the algorithms described in Section 11.2.1 to compute D_2^s, D_2^*, and similar variants based on $p(W)$, using the bidirectional BWT index of S and T, and state their complexity.

11.15 Show how to implement the computations described in Insight 11.3 using the bidirectional BWT index of $S \in [1..\sigma]^n$. More precisely, do the following.

(a) Show how to compute the number of distinct k-mers that appear at least twice (repeating k-mers) in time $O(n \log \sigma)$ and space $O(n \log \sigma)$ bits.
(b) Suppose that we want to compute the number of repeating k-mers for all values of k in a given range $[k_1..k_2]$. A naive extension of the algorithm in (a) would take time $O((k_1 - k_2)n \log \sigma)$ and it would use an additional $O((k_1 - k_2)\log n)$ bits of space to store the result. Show how to improve this running time to $O(n \log \sigma)$, using the same space.
(c) Show how to compute the Kullback–Leibler divergence described in Insight 11.3 simultaneously for all values of k in $[k_1..k_2]$, in time $O(n \log \sigma)$ and an additional $O(k_1 - k_2)$ computer words of space to store the result.

11.16 Assume that vectors $\mathbf{S}$ and $\mathbf{T}$ have only two dimensions. Show that Equation (11.1) is indeed the cosine of the angle between $\mathbf{S}$ and $\mathbf{T}$.

11.17 Write a program that computes all kernels and complexity measures described in Section 11.2.1, given the suffix tree of the only input string, or the generalized suffix tree of the two input strings.

11.18 Adapt the algorithm in Section 11.2.2 to compute the substring kernel with Markovian corrections in $O(n \log \sigma)$ time and space, where n is the sum of the lengths of the input strings.

11.19 Given strings W and S, and characters a and b on an alphabet $\Sigma = [1..\sigma]$, consider the following expected probability of observing aWb in S, analogous to Equation (11.2):

$$\tilde{p}_{W,S}(a,b) = \frac{p_S(aW) \cdot p_S(Wb)}{p_S(W)}.$$

Moreover, consider the Kullback–Leibler divergence between the observed and expected distribution of $\tilde{p}_{W,S}$ over pairs of characters (a,b):

$$\mathsf{KL}_S(W) = \sum_{(a,b)\in\Sigma\times\Sigma} p_{W,S}(a,b) \cdot \ln\left(\frac{p_{W,S}(a,b)}{\tilde{p}_{W,S}(a,b)}\right),$$

where we set $\ln(p_{W,S}(a,b)/\tilde{p}_{W,S}(a,b)) = 0$ whenever $\tilde{p}_{W,S}(a,b) = 0$. Given strings S and T, let $\mathbf{S}$ and $\mathbf{T}$ be infinite vectors indexed by all strings on alphabet $[1..\sigma]$, such that $\mathbf{S}[W] = \mathsf{KL}_S(W)$ and $\mathbf{T}[W] = \mathsf{KL}_T(W)$. Describe an algorithm to compute the cosine of vectors $\mathbf{S}$ and $\mathbf{T}$ as defined in Equation (11.1).

11.20 Recall that the *shortest unique substring array* $\mathsf{SUS}_S[1..n]$ of a string $S \in [1..\sigma]^n$ is such that $\mathsf{SUS}_S[i]$ is the length of the shortest substring that starts at position i in S and that occurs exactly once in S.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

i. Adapt to SUS_S the left-to-right and right-to-left algorithms described in Section 11.2.3 for computing matching statistics.
ii. Describe how to compute SUS_S by an $O(n)$ bottom-up navigation of the suffix tree of S.

11.21 Show how to compute $\sum_{j=|\ell(u)|+1}^{|W|} \alpha(W[1..j])$ in constant time for all the weighting schemes described in Section 11.2.3. More generally, show that, if $\alpha(W)$ depends just on the length of W rather than on its characters, and if $\alpha(W) = 0$ for $|W|$ bigger than some threshold τ, we can access $\sum_{j=|\ell(u)|+1}^{|W|} \alpha(W[1..j])$ in constant time after an $O(\tau)$-time and $O(\tau \log \tau)$-space precomputation.

11.22 Let S and T be strings, and let $D = \{d_1, d_2, \ldots, d_k\}$ be a fixed set of strings with weights $\{w_1, w_2, \ldots, w_k\}$. Show how to compute the kernel $\kappa(S, T)$ in Equation (11.4) restricted to the strings in D, using matching statistics and with the same time complexity as that of Theorem 11.22.

11.23 Given strings S and T and a window size $w \in [1..|S| - 1]$, describe an algorithm that computes the kernel $\kappa(S[i..i + w - 1], T)$ in Equation (11.4) for all $i \in [1..|S| - w + 1]$, performing fewer than m operations per position. *Hint*. Adapt the approach in Theorem 11.22.

11.24 Consider the suffix-link tree SLT_T of a string T, augmented with implicit Weiner links and their destinations. Describe an algorithm that adds to the suffix tree $\mathsf{ST}_{\underline{T}}$ a node for every label W of a node in SLT_T (if string W is not already the label of a node in $\mathsf{ST}_{\underline{T}}$), and that adds to every node of this augmented suffix tree a pointer to its closest ancestor labeled by a string in SLT_T. The relationship between SLT_T and $\mathsf{ST}_{\underline{T}}$ is further explored in Exercise 8.19.

11.25 Given two k-mers V and W on alphabet $[1..\sigma]$, recall that $D_{\mathsf{H}}(V, W)$ is the Hamming distance between V and W, and that $B(W, m)$ is the subset of $[1..\sigma]^k$ that lies at Hamming distance at most m from W. Give a closed-form expression for the size of $B(V, m) \cap B(W, m)$ for every possible value of $D_{\mathsf{H}}(V, W)$ when $m = 2$.

11.26 Given a string S on alphabet $[1..\sigma]$, and given a string W of length k, let $f_S(W, m)$ be the number of substrings of S of length $k + m$ that contain W as a *subsequence*. Consider the following *gapped kernel* between two strings S and T:

$$\kappa(S, T, k, m) = \sum_{W \in [1..\sigma]^k} f_S(W, m) \cdot f_T(W, m).$$

Adapt the approach described in Section 11.2.4 to compute the gapped kernel, and state its complexity.

11.27 In the *neighborhood kernel*, the similarity of two strings S and T is expressed in terms of the similarities of their *neighborhoods* $N(S)$ and $N(T)$, where $N(S)$ contains a given set of strings that the user believes to be similar or related to S. The neighborhood

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press

kernel is defined as

$$\kappa_N(S,T) = \sum_{U \in N(S)} \sum_{V \in N(T)} \kappa(U,V),$$

where $\kappa(U,V)$ is a given kernel. Describe how to adapt the approaches described in Section 11.2.4 to compute the neighborhood kernel.

11.28 Prove that $C(ST) + C(U) \leq C(SU) + C(TU)$, by using the symmetry of C.

11.29 Complete the proof of Theorem 11.31 with the two missing cases, using the distributivity of C.

https://doi.org/10.1017/CBO9781139940023.013 Published online by Cambridge University Press