16 Metagenomics

Assume that a drop of seawater contains cells from n distinct species $\mathcal{S} = \{S^1, S^2, \ldots, S^n\}$. Denote the genomes of such species by $\mathcal{G} = \{G^1, G^2, \ldots, G^n\}$, and let $\mathbf{F}[1..n]$ be a vector that contains the number of cells of each species. We use the related vector $\mathbf{P}[i]$ to denote *relative frequencies*: in other words, $\mathbf{P}[i]$ equals $\mathbf{F}[i]$ divided by the total number of cells in the drop of water. In practice the distribution $\mathbf{P}$ is heavy-tailed, with few species at high frequency and many species at low frequency. We call the triplet $(\mathcal{S}, \mathcal{G}, \mathbf{P})$ the *environment*. In most practical cases n is large, we know just a small fraction of $\mathcal{S}$ and $\mathcal{G}$, and we completely ignore $\mathbf{P}$ and $\mathbf{F}$. A *metagenomic sample* is a collection of r strings $\mathcal{R} = \{R^1, R^2, \ldots, R^r\}$, where each R^i, called a *read*, is a substring of a genome in $\mathcal{G}$, possibly perturbed by sequencing errors. Methods and tools for the computational analysis of metagenomic samples are still under active development, and some of the questions raised by this type of heterogeneous, large-scale data remain unresolved. In this chapter we focus on a subset of key problems with a clear algorithmic formulation.

Throughout the chapter we assume that, except for a small variance, all reads in a sample have the same length ℓ, which is significantly smaller than the length of any genome in $\mathcal{G}$. For simplicity, we assume that the process that generated $\mathcal{R}$ from the environment sampled a substring of length ℓ with equal probability from any position of the genome of any cell: in other words, the probability that a read comes from genome G^i is $|G^i| \cdot \mathbf{F}[i] / \sum_{j=1}^{n} |G^j| \cdot \mathbf{F}[j]$. We call the average number of reads that cover a position in G^i, or equivalently the number of reads in $\mathcal{R}$ that were sampled from a cell with genome G^i, multiplied by $\ell/|G^i|$, the *coverage* of a genome $G^i \in \mathcal{G}$.

In practice just a small fraction of all existing genomes has been sequenced: we might know some genomes just as a set of contigs (see Section 13.2), or just as an even smaller set of *genes*. In some applications we are more interested in estimating the biological functions performed by the species in an environment, rather than the relative frequency of the species themselves: for this purpose we might use precomputed *gene families*, collections of sequenced genes with similar functions coming from known species. Such families are typically annotated with a reference alignment or with a hidden Markov model that describes the common features of all sequences in a family (see Section 6.6 and Chapter 7). In this chapter we denote by $\mathcal{D}$ any precomputed database that is used by an algorithm, leaving its details to the context.

We also denote by $\mathcal{T}$ a known *reference taxonomy*, that is, a tree whose leaves are all known species, and whose (possibly unary) internal nodes, called *taxa*, are higher-order groups of species based on evolutionary history or on shared biological features. The depth of a taxon in $\mathcal{T}$ is called its *taxonomic rank*. When it is hard to estimate the composition of a sample at the species level, algorithms focus on immediately smaller ranks, called *genus* and *family*.

16.1 Species estimation

A key problem in metagenome analysis consists in reconstructing the *taxonomic composition* of a sample, or in other words in estimating $\mathcal{S}$ and $\mathbf{P}$ from $\mathcal{R}$. Knowing which taxa are present in a sample, and in which proportion, can help one to study the effects of a drug, of a pathogen, or of a disease on the microbes that live in the human gut, for example. The taxonomic composition can also serve as a succinct fingerprint to compare, classify, and retrieve large collections of metagenomic samples. The methods described in this section can be used almost verbatim with databases of protein families, to estimate the biochemical processes that are at work in an environment, rather than its taxonomic composition.

16.1.1 Single-read methods

Assume that all genomes in $\mathcal{G}$ are stored in a reference database $\mathcal{D}$. Perhaps the most natural way of estimating $\mathcal{S}$ and $\mathbf{P}$ from $\mathcal{R}$ consists in finding the substring W^i of a genome in $\mathcal{D}$ that best matches each read $R^i \in \mathcal{R}$ taken in isolation, for example by using the read-mapping techniques described in Chapter 10. In this section we assume that every substring W of a genome in $\mathcal{D}$ that matches a read R^i has a corresponding *matching score* (or, symmetrically, a *matching cost*, for example the edit distance between R^i and W) that quantifies the quality of the alignment between R^i and W. The search for a best match can be naturally performed in parallel for each read, or for each genome in $\mathcal{D}$. If R^i is long enough, we could first compare its k-mer composition with the k-mer composition of the genomes or of the gene families in $\mathcal{D}$, and then match R^i just to the portions of the database with similar enough k-mer composition (see Section 11.2 for a sampler of similarity measures based on k-mers).

A read R^i can approximately match more than one genome in $\mathcal{D}$, or even the same genome at different positions: let $\mathcal{W}^i$ denote the set of all substrings of $\mathcal{D}$ that match R^i with a high enough score. If $|\mathcal{W}^i| > 1$, it is common practice to assign R^i to the *lowest common ancestor* in $\mathcal{T}$ of all elements in $\mathcal{W}^i$: this effectively builds an approximation of $\mathcal{S}$ and $\mathbf{P}$ in which species and relative frequencies are grouped by $\mathcal{T}$, possibly at different depths. This strategy can be refined by re-mapping the best match W^i to $\mathcal{W}^i$ itself, and by assigning R^i to the lowest common ancestor *of the matches of* W^i *in* $\mathcal{W}^i$ whose score is higher than the matching score between W^i and R^i. Strategies based on the least common ancestor, however, tend to assign reads to taxa with low rank in $\mathcal{T}$, thereby limiting the accuracy of the estimate, and they discard reads with many high-scoring matches when the corresponding lowest common ancestor is close to the root of $\mathcal{T}$.

Marker-based methods

Rather than using entire genomes as a reference database, we could employ a precomputed set of *markers* that are known to uniquely characterize a node in $\mathcal{T}$.

DEFINITION 16.1 *A* marker *for a node $v \in \mathcal{T}$ is a substring that occurs at most τ times in every genome in v, and that does not occur in any genome not in v.*

We can compute such markers of maximal length by adapting the document counting algorithm described in Section 8.4.2, or alternatively by using the matching statistics and the distinguishing statistics vectors described in Section 11.2.3 (see Exercises 16.11 and 16.12). If we restrict markers to having a specific length k, we can compute all of them in small space, as follows.

LEMMA 16.2 *Let $\mathcal{S} = \{S^1, S^2, \ldots, S^n\}$ and $\mathcal{T} = \{T^1, T^2, \ldots, T^m\}$ be two sets of strings on alphabet $[1..\sigma]$. Given the bidirectional BWT index of S^i for all $i \in [1..n]$, the bidirectional BWT index of T^j for all $j \in [1..m]$, and two integers $k > 1$ and $\tau \geq 1$, we can compute all k-mers that occur between two and τ times in every string of $\mathcal{S}$, and that do not occur in any string of $\mathcal{T}$, in $O((x+y)\log\sigma \cdot (|\mathcal{S}| + |\mathcal{T}|))$ time and $3x + o(x) + y + O(\sigma \log^2 y)$ bits of working space, where $x = \min\{|S^i| : i \in [1..n]\}$ and $y = \max\{|X| : X \in \mathcal{S} \cup \mathcal{T}\}$.*

Proof Let S^* be a string of minimum length in $\mathcal{S}$. We build a bitvector `intervals`$[1..|S^*|+1]$ such that `intervals`$[i] = 1$ if and only if i is the starting position of a k-mer interval in $\mathsf{BWT}_{S^*\#}$, and `intervals`$[i] = 0$ otherwise. Recall from Section 9.7.3 that this vector can be built from the bidirectional BWT index of S^* in $O(|S^*|\log\sigma)$ time, by enumerating the internal nodes of the suffix tree of S^*. We also initialize to zero a bitvector `intersection`$[1..|S^*|+1]$, and to one a bitvector `found`$[1..|S^*|+1]$. Then, we scan `intervals` sequentially, and we set `intersection`$[p] = 1$ for every starting position p of an interval of size at most τ in `intervals`. We index `intervals` to support `rank` queries in constant time, as described in Section 3.2.

Then, we consider every other string $S^i \in \mathcal{S} \setminus \{S^*\}$. First, we build a bitvector `intervals`$^i[1..|S^i|+1]$ marking the intervals of all k-mers in $\mathsf{BWT}_{S^i\#}$. Then, we use the bidirectional BWT index of S^* and the bidirectional BWT index of S^i to enumerate the internal nodes of their generalized suffix tree, as described in Algorithm 11.3. Assume that we reach a node v of the generalized suffix tree whose depth is at least k: let $\overleftrightarrow{v}_i$ and $\overleftrightarrow{v}_*$ be the intervals of v in $\mathsf{BWT}_{S^i\#}$ and in $\mathsf{BWT}_{S^*\#}$, respectively, and let $\overleftrightarrow{W}_i$ and $\overleftrightarrow{W}_*$ be the intervals of the length-k prefix W of $\ell(u)$ in $\mathsf{BWT}_{S^i\#}$ and in $\mathsf{BWT}_{S^*\#}$, respectively. If $\overleftrightarrow{v}_i$ coincides with the interval of a k-mer in $\mathsf{BWT}_{S^i\#}$, we compute $\overleftrightarrow{W}_*$ in constant time from $\overleftrightarrow{v}_*$ and `intervals`, and we set `found`$[p] = 1$, where p is the starting position of $\overleftrightarrow{W}_*$ in $\mathsf{BWT}_{S^*\#}$. At the end of this process, we set `intersection`$[i] = 0$ for all $i \in [1..|S^*|+1]$ such that `intersection`$[i] = 1$ and `found` $= 0$. A similar algorithm can be applied to unmark from `intersection` the starting position of all k-mers that occur in a string of $\mathcal{T}$.

Then, we use again the bidirectional BWT index of S^* and the bidirectional BWT index of S^i to enumerate the internal nodes of their generalized suffix tree. Assume that we reach a node v of the generalized suffix tree, at depth at least k, such that $\vec{v}_*$ coincides with the interval of a k-mer in $\mathsf{BWT}_{S^*\#}$, and such that $\mathtt{intersection}[p] = 1$, where p is the starting position of $\vec{v}_*$. Then we set $\mathtt{intersection}[p] = 0$ if $\vec{v}_i$ coincides with the interval of a k-mer in $\mathsf{BWT}_{T^i\#}$, and if $|\vec{v}_i| > \tau$. At the end of this process, the bits set to one in `intersection` mark the starting positions of the intervals of the distinct k-mers of S^* that occur between two and τ times in every string of $\mathcal{S}$.

Once all of the strings in $\mathcal{S}$ and $\mathcal{T}$ have been processed in this way, we invert $\mathsf{BWT}_{S^*\#}$, and we return the length-k prefix of the current suffix of $S^*\#$ whenever we reach a marked position in `intersection`. □

It is easy to generalize Lemma 16.2 to k-mers that occur between *one* and τ times.

For concreteness, assume that markers have been precomputed for every genome in $\mathcal{G}$. Recall that the probability that a read in $\mathcal{R}$ is sampled from a genome G^i is $(|G^i| \cdot \mathbf{F}[i])/(\sum_{j=1}^{n} |G^j| \cdot \mathbf{F}[j])$. The coverage of G^i, or equivalently the number of reads that cover a position of G^i, is thus

$$C(i) = |\mathcal{R}|\ell \cdot \frac{\mathbf{F}[i]}{\sum_{j=1}^{n} |G^j| \cdot \mathbf{F}[j]}$$

and the sum of all coverages is

$$\sum_{i=1}^{n} C(i) = |\mathcal{R}|\ell \cdot \frac{\sum_{i=1}^{n} \mathbf{F}[i]}{\sum_{j=1}^{n} |G^j| \cdot \mathbf{F}[j]}.$$

It follows that $\mathbf{P}[i] = C(i)/\sum_{i=1}^{n} C(i)$. Let M^i be the concatenation of all markers of genome G^i, and let $\mathcal{R}^i$ be the set of reads in $\mathcal{R}$ whose best match is located inside a marker of G^i. We can estimate $C(i)$ by $|\mathcal{R}^i|\ell/|M^i|$ for every G^i, and thus derive an estimate of $\mathbf{P}[i]$.

In practice aligning reads to markers can be significantly faster than aligning reads to entire genomes, since the set of markers is significantly smaller than the set of genomes, and the probability that a read matches a marker is lower than the probability that a read matches a substring of a genome. This method allows one also to detect taxa that are not in $\mathcal{T}$, since the difference between the relative frequency of a taxon v and the sum of the relative frequencies of all children of v in $\mathcal{T}$ can be assigned to an unknown child below v.

16.1.2 Multi-read and coverage-sensitive methods

Rather than assigning each read to a taxon in isolation, we could combine the taxonomic assignments of multiple reads to increase both the number of classified reads and the accuracy of the estimate. For example, we could first use the set of all mappings of all reads to reference genomes in $\mathcal{D}$ to determine *which taxa in $\mathcal{T}$ are present in the sample*, and only afterwards try to assign reads to taxa. Specifically, we could compute the smallest set of taxa that can annotate all reads, at each taxonomic rank i of $\mathcal{T}$: this is an instance of the set cover problem described in Section 14.1, in which the taxa at rank i are sets, and a taxon v *covers* a read R^j if R^j has a high-scoring match in some genomes that belong to v. Then, read R^j can be assigned to a largest set that covers it, or it can be declared unclassified if the assigned taxon at rank i is not a child of the assigned taxon at rank $i - 1$.

Another way of combining information from multiple reads consists in assembling reads into contigs (see Section 13.2), and then mapping such long contigs to the reference genomes in $\mathcal{D}$. Longer contigs reduce the number of false matches, and potentially allow one to annotate reads that could not be annotated when mapped to the database in isolation. Contigs could be preliminarily clustered according to their k-mer composition before being matched to the database: all contigs in the same cluster X could then be assigned to the lowest common ancestor of the best matches of all contigs in X. Similar strategies can be applied to clusters of reads built without using an assembler, as described in Section 16.2.

A third way of using global information consists in exploiting the assumption that *the probability of sampling a read from a given position in a given genome is the same for all positions in the genome*. We call a method that exploits this assumption *coverage-sensitive*. In what follows, we denote by $\mathcal{D}' = \{D^1, \dots, D^m\}$ the subset of $\mathcal{D}$ that contains all genomes with a high-scoring match to at least one read in $\mathcal{R}$. We want to select exactly one mapping location for each read (or to declare the read unmapped), so that the resulting coverage is as uniform as possible along the sequence of every genome in $\mathcal{D}'$, and so that the sum of all matching costs is as small as possible: this provides an estimate of the number of reads c_i sampled from each genome $D^i \in \mathcal{D}'$, which can be immediately converted to its relative frequency $\mathbf{P}[i]$. It is natural to express this problem with the formalism of minimum-cost flows, as follows.

Mapping reads using candidate abundances

Assume first that we already have an estimate of the number of reads $c_1, \dots, c_m$ sampled from each genome in $\mathcal{D}'$, which we call the *candidate abundance*. We will deal later with the problem of finding an optimal set of such abundances. If each genome D^i has uniform coverage, then its uniform *candidate coverage* is $c_i/(|D^i| - \ell + 1)$. In order to model the uniformity constraint, we partition every genome D^i into substrings of a fixed length L, which we will refer to as *chunks*. Denote by s_i the number of chunks that each genome D^i is partitioned into. We construct a bipartite graph $G = (A \cup B, E)$, such that the vertices of A correspond to reads, and the vertices of B correspond to the chunks of all genomes $D^1, \dots, D^m$. Specifically, for every chunk j of genome D^i, we introduce a vertex $y_{i,j}$, and we add an edge between a read $x \in A$ and vertex $y_{i,j} \in B$ if there is a match of read x inside chunk j of genome D^i. This edge is assigned the cost of the match, which we denote here by $c(x, y_{i,j})$. The uniformity assumption can now be modeled by requiring that every genome chunk receives close to c_i/s_i read mappings. In order to model the fact that reads can originate from unknown species whose genome is not present in $\mathcal{D}'$, we introduce an "unknown" vertex z in B, with edges from every read vertex $x \in A$ to z, and with an appropriately initialized cost $c(x, z)$.

In the problem of coverage-sensitive abundance evaluation stated below, the task is to select exactly one edge for every vertex $x \in A$ (a mapping of the corresponding read), either to a chunk of some genome D^i, or to the unknown vertex z, which at the same time minimizes the sum, over all chunks of every genome D^i, of the absolute difference between c_i/s_i and the number of mappings it receives, and also minimizes the sum of all selected edge costs. We can combine these two optimization criteria into a single objective function by receiving in the input also a parameter $\alpha \in (0, 1)$, which controls the relative contribution of the two objectives. Given a set M of edges in a graph G, recall from page 53 that we denote by $d_M(x)$ the number of edges of M incident to a vertex x of G. Formally, we have the following problem.

Problem 16.1 Coverage-sensitive mapping and abundance evaluation

Given a bipartite graph $G = (A \cup B, E)$, where A is a set of n reads and $B = \{y_{1,1}, \ldots, y_{1,s_1}, \ldots, y_{m,1}, \ldots, y_{m,s_m}\} \cup \{z\}$ is a set of genome chunks; a cost function $c : E \to \mathbb{Q}$; a constant $\alpha \in (0, 1)$; and a candidate tuple $c_1, \ldots, c_m$ of reads per genome, find a many-to-one matching M that satisfies the condition

- for every $x \in A$, $d_M(x) = 1$ (that is, every read is covered by exactly one edge of M),

and minimizes the following quantity:

$$(1-\alpha)\sum_{(x,y)\in M} c(x,y) + \alpha \sum_{i=1}^{m}\sum_{j=1}^{s_i}\left|\frac{c_i}{s_i} - d_M(y_{i,j})\right|. \qquad (16.1)$$

This problem can be solved in polynomial time by a reduction to a minimum-cost flow problem, as was done for the problem of many-to-one matching with optimal residual load factors from Section 5.3.3. The only difference here is the introduction of the vertex z, which does not have any required load factor. Exercise 16.1 asks the reader to work out the details of this reduction.

Finding optimal abundances

We still need to find the *optimal* tuple $c_1, \ldots, c_N$ of abundances which minimizes (16.1) in Problem 16.1 over all possible tuples of abundances. We extend the formulation of Problem 16.1 by requiring to find the optimal such abundances as well.

Problem 16.2 Coverage-sensitive mapping and abundance estimation

Given a bipartite graph $G = (A \cup B, E)$, where A is a set of n reads, and $B = \{y_{1,1}, \ldots, y_{1,s_1}, \ldots, y_{m,1}, \ldots, y_{m,s_m}\} \cup \{z\}$ is a set of genome chunks; a cost function $c : E \to \mathbb{Q}$; and constant $\alpha \in (0, 1)$, find a many-to-one matching M that satisfies the condition

- for every $x \in A$, $d_M(x) = 1$ (that is, every read is covered by exactly one edge of M), and find a tuple of abundances $c_1, \ldots, c_N$ that minimizes the following quantity:

$$(1-\alpha)\sum_{(x,y)\in M} c(x,y) + \alpha \sum_{i=1}^{m} \sum_{j=1}^{s_i} \left| \frac{c_i}{s_i} - d_M(y_{i,j}) \right|. \tag{16.2}$$

Unfortunately, this problem is NP-hard, as shown in Theorem 16.3. However, in practice, since Problem 16.1 is solvable in polynomial time, we can use ad-hoc algorithms to guess different candidate tuples of abundances, and then we can evaluate their performance using Problem 16.1.

THEOREM 16.3 *Problem 16.2 is NP-hard.*

Proof We reduce from the problem of exact cover with 3-sets (X3C). In this problem, we are given a collection $\mathcal{S}$ of 3-element subsets $S_1, \ldots, S_m$ of a set $U = \{1, \ldots, n\}$, and we are required to decide whether there is a subset $\mathcal{C} \subseteq \mathcal{S}$, such that every element of U belongs to exactly one $S_i \in \mathcal{C}$.

Given an instance of problem X3C, we construct the bipartite graph $G = (A \cup B, E)$, where $A = U = \{1, \ldots, n\}$ and B corresponds to $\mathcal{S}$, in the following sense:

- $s_i = 3$, for every $i \in \{1, \ldots, m\}$;
- for every $S_j = \{i_1 < i_2 < i_3\}$, we add to B the three vertices $y_{j,1}, y_{j,2}, y_{j,3}$ and the edges $\{i_1, y_{j,1}\}$, $\{i_2, y_{j,2}\}$, $\{i_3, y_{j,3}\}$, each with cost 0.

For completeness, we also add vertex z to B, and edges of some positive cost between it and every vertex of A.

We now show that, for any $\alpha \in (0,1)$, an instance for problem X3C is a "yes" instance if and only if Problem 16.2 admits on this input a many-to-one matching M of cost 0. Observe that, in any solution M of cost 0, the genome abundances are either 0 or 3, and that vertex z has no incident edges in M.

For the forward implication, let $\mathcal{C}$ be an exact cover with 3-sets. We assign the abundances $c_1, \ldots, c_m$ as follows:

$$c_j = \begin{cases} 3, & \text{if } S_j \in \mathcal{C}, \\ 0, & \text{if } S_j \notin \mathcal{C}; \end{cases}$$

and we construct M as containing, for every $S_j = \{i_1 < i_2 < i_3\} \in \mathcal{C}$, the three edges $\{i_1, y_{j,1}\}$, $\{i_2, y_{j,2}\}$, $\{i_3, y_{j,3}\}$. Clearly, this M gives a solution of cost 0 to Problem 16.2.

Vice versa, if Problem 16.2 admits a solution M of cost 0, in which the genome coverages are either 0 or 3 and z has no incident edges, then we can construct an exact cover with 3-sets $\mathcal{C}$ by taking

$$\mathcal{C} = \{S_i \mid d_M(y_{i,1}) = d_M(y_{i,2}) = d_M(y_{i,3}) = 1\}.$$

□

16.2 Read clustering

When reads in $\mathcal{R}$ cannot be successfully mapped to known genomes, we might want to assemble them into the corresponding, unknown source genomes, for example by using the algorithms described in Chapter 13. This corresponds to estimating just $\mathcal{G}$ and **P**, but not $\mathcal{S}$. However, the size of metagenomic samples, the large number of species they contain, and the sequence similarity of such species, make assembly a slow and inaccurate process in practice. It might thus be useful to preliminarily group reads together on the basis of their k-mer composition: the resulting clusters loosely correspond to long strings that occur in one or in multiple genomes of $\mathcal{G}$, and in the best case to entire genomes. Assembly can then be carried out inside each cluster, possibly in parallel. The number of clusters can also be used to estimate **P** and derived measures of the biodiversity of an environment, and the k-mer composition of a cluster can be used to position it approximately inside a known taxonomy, without assembling its reads. Like the frequency of species, the number and k-mer composition of clusters can also be used as a succinct fingerprint to compare and classify large collections of metagenomic samples.

Clustering reads becomes more difficult as the number of species increases, and as the frequency vector **P** diverges from a uniform distribution. To improve accuracy, it is becoming more common to combine multiple metagenomic samples that are believed to contain approximately the same species, possibly with different relative frequencies. The input to this problem is then even larger than the already massive metagenomic samples taken in isolation. In this section we describe a space-efficient clustering pipeline that is based on the bidirectional BWT index described in Section 9.4, where R is the concatenation of all reads in $\mathcal{R}$, each terminated by a distinct binary string on the artificial characters $\{\#_0, \#_1\} \notin [1..\sigma]$. The main steps of the pipeline are summarized in Figure 16.1.

16.2.1 Filtering reads from low-frequency species

As mentioned at the beginning of this chapter, the distribution of relative frequencies **P** is heavy-tailed, with few species at high frequency and many species at low frequency. In practice reads from low-frequency species tend to degrade the quality of the clusters of high-frequency species: a preliminary step consists thus in isolating reads from low-frequency species and processing them in a separate phase. This has the additional advantages of removing reads with many errors and of reducing the size of the input of each phase.

Specifically, a read is filtered out if and only if *all its k_1-mers occur fewer than τ_1 times in $\mathcal{R}$*, where k_1 and τ_1 are user-defined parameters that depend on the error rate, coverage, and read length.

LEMMA 16.4 *Given the bidirectional BWT index of R, we can detect in $O(|R| \log \sigma)$ time and in $2|R| + o(|R|) + \sigma \log^2 |R|$ bits of working space all of the reads in R that contain at least one k_1-mer that occurs at least $\tau_1 \geq 2$ times in R.*

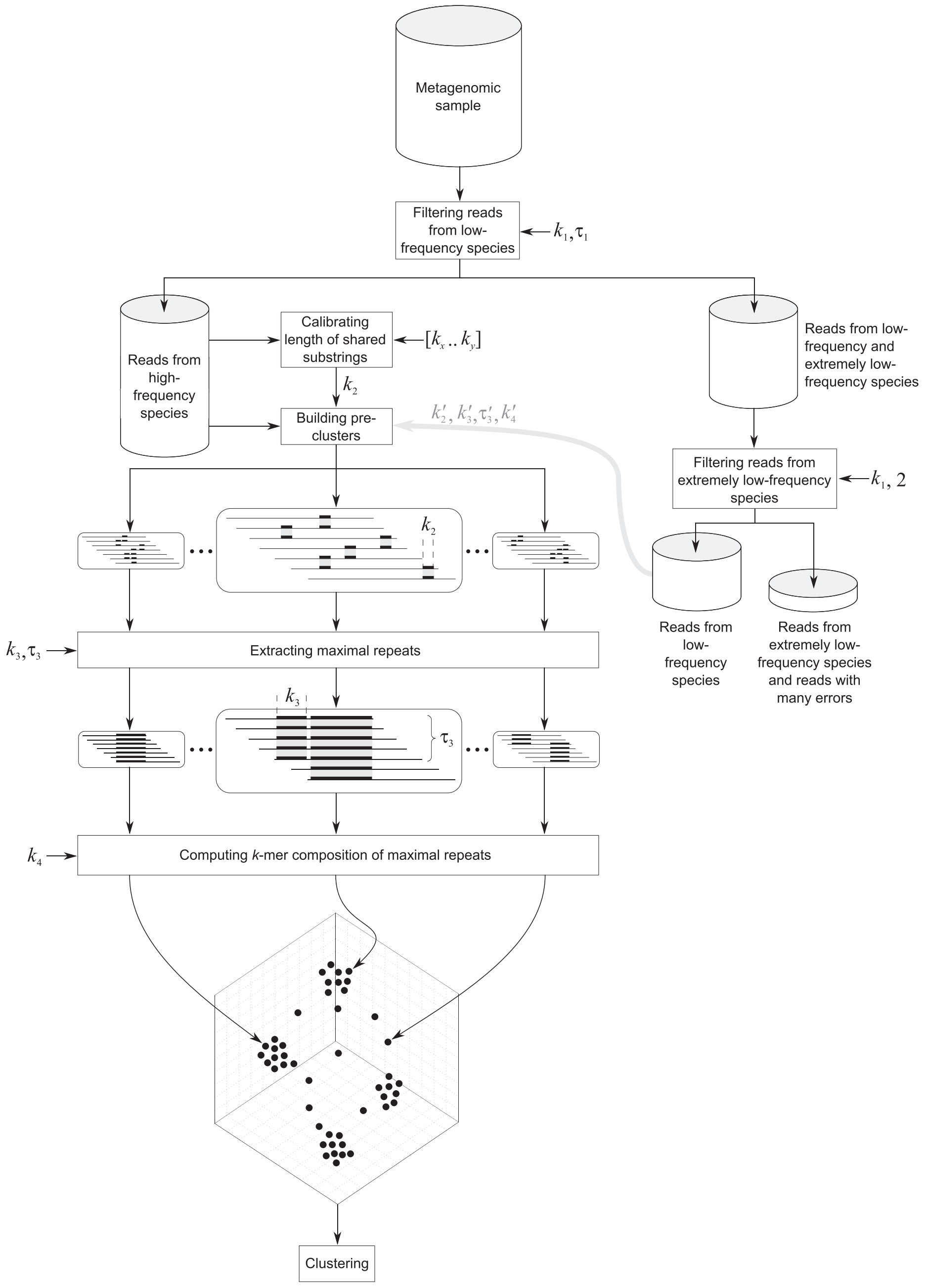
Metagenomic sample
Filtering reads from low-frequency species
k_1, τ_1
Reads from high-frequency species
Calibrating length of shared substrings
$[k_x .. k_y]$
k_2
Building pre-clusters
$k'_2, k'_3, \tau'_3, k'_4$
Reads from low-frequency and extremely low-frequency species
Filtering reads from extremely low-frequency species
k_1, 2
k_2
Reads from low-frequency species
Reads from extremely low-frequency species and reads with many errors
k_3, τ_3
Extracting maximal repeats
k_3
τ_3
k_4
Computing *k*-mer composition of maximal repeats
Clustering

Figure 16.1 The main steps for clustering metagenomic reads. See Section 16.2 for a detailed description of each step and parameter. The light gray arc indicates that reads from low-frequency species can be processed with the same pipeline as reads from high-frequency species, but using different parameters.

Proof We build a bitvector of intervals $\texttt{intervals}[1..|R|]$ such that $\texttt{intervals}[i] = 1$ if and only if i is the starting or the ending position of the interval of a k_1-mer in BWT_R, and $\texttt{intervals}[i] = 0$ otherwise. Recall from Section 9.7.3 that by enumerating the internal nodes of the suffix tree of R in $O(|R| \log \sigma)$ time, using the bidirectional BWT index of R, we can build a bitvector $\texttt{first}_{k_1}$ that is zero everywhere, except at the first position of the interval of every k_1-mer in BWT_R. We can easily derive $\texttt{intervals}$ by a linear scan of $\texttt{first}_{k_1}$. During this scan, we avoid marking the intervals of k_1-mers that are smaller than τ_1. Finally, we preprocess $\texttt{intervals}$ in linear time so that it supports $\texttt{rank}$ queries in constant time (see Section 3.2). We then build a temporary bitvector $\texttt{keep}[1..r]$ initialized to zeros, with a bit for each read in $\mathcal{R}$, and we invert BWT_R: at each step of the inversion, we know a position i in R, the read q this position belongs to, and the corresponding position j in BWT_R. If j belongs to an interval in $\texttt{intervals}$, we set $\texttt{keep}[q] = 1$. □

Note that, during the enumeration of the internal nodes of the suffix tree of R, we can waive the trick described in Algorithm 9.3 to keep the depth of the stack bounded by $\log |R|$, and we can instead perform a standard pre-order traversal of the suffix-link tree of R: indeed, the depth of the stack is bounded by the read length ℓ, which is a small constant in practice.

From now on, we assume that the reads from low-frequency species are stored on disk, that R denotes the concatenation of all reads from high-frequency species only, and that the bidirectional BWT index is built on this new version of R. Note that, rather than computing the Burrows–Wheeler transforms of the new R and $\underline{R}$ from scratch, we could derive them from their previous versions: see Exercise 16.6.

16.2.2 Initializing clusters

Let W^i_1 and W^i_2 be two, sufficiently long substrings of the same genome G^i, and let W^j be a substring of a different genome G^j. It is typically the case that the k-mer composition of W^i_1 is more similar to the k-mer composition of W^i_2 than to the k-mer composition of W^j for suitably small k. If the reads are long enough, we can thus compute their k-mer compositions and use this directly to build clusters that approximate genomes. If the reads are short, however, we cannot estimate their k-mer composition reliably: a possible workaround consists in grouping similar reads into *pre-clusters*, estimating the k-mer composition of such pre-clusters, and iteratively merging pre-clusters according to their k-mer compositions.

Perhaps the most natural criterion to create pre-clusters consists in merging two reads if they share a sufficiently long substring: this makes pre-clusters loosely correspond to unassembled contigs (see Section 13.2), or to substrings of a genome. Creating this kind of pre-cluster has the additional advantage of assigning approximately the same number of pre-clusters to species with very different frequencies. Indeed, consider two species $\{G^i, G^j\} \subseteq \mathcal{G}$ such that $\mathbf{P}[i]$ is very different from $\mathbf{P}[j]$: since pre-clusters loosely correspond to substrings of a genome, we expect that all the pre-clusters associated with G^i contain approximately the same number of reads c_i, and that all the pre-clusters associated with G^j contain approximately the same number of reads c_j, with c_i very different from c_j. However, the number of pre-clusters associated with each genome should depend only on its length, and the variation in length among different species in the same metagenomic sample is significantly lower than the variation in their frequency.

But how can we choose the length of the shared substrings to be used to create pre-clusters? In practice it is common to start from a range of *candidate lengths*, say $[k_x..k_y]$. Such a range could have been estimated from known genomes, so that a substring of length $k \in [k_x..k_y]$ occurs with high probability at most once in every known genome. Alternatively, the range could have been computed from some estimate of the error rate and coverage, to minimize the probability that two reads from different genomes share a substring of length inside the range. Note that this probability decreases as k increases, but setting k too large might prevent reads from the same genome being clustered together, because of sequencing errors. We thus want to derive, *from the read set $\mathcal{R}$ itself*, the maximum $k_2 \in [k_x..k_y]$ such that *the majority of distinct k_2-mers of R will occur at least twice in R*: in other words, most such k_2-mers are not likely to contain sequencing errors. This can be done in a single traversal of the internal nodes of the suffix tree of R, using the techniques described in Section 11.2.1: see Exercise 11.15.

Once we have calibrated k_2, we can merge two reads if they share at least one k_2-mer:

LEMMA 16.5 *Given the bidirectional BWT index of R and an integer k_2, we can build in $O(|R| \log \sigma)$ time and in* $\max\{|R| + \sigma \log^2 |R|, 2|R|, |R| + o(|R|) + K \log r, r \log r\}$ *bits of additional space all pre-clusters of R induced by k_2-mers, where K is the number of distinct repeating k_2-mers of R.*

Proof We compute again the bitvector $\mathtt{first}_{k_2}$ using Lemma 9.22 in Section 9.7.3, and we compute from $\mathtt{first}_{k_2}$ another bitvector, `intervals`, which marks the first and the last position of all the disjoint k_2-mer intervals of size at least two in BWT_R. Note that we can avoid marking in `intervals` the boundaries of the intervals of other k_2-mers as well. Indeed, consider the interval of k_2-mer W in BWT_R, and extend W to the left by one character in all possible ways: we can discard any of the resulting intervals if it happens to be itself the interval of a k_2-mer. Such a test can be performed during a linear scan of BWT_R: see Exercise 16.7. We then index `intervals` to support `rank` queries in constant time (see Section 3.2). Let K be the number of distinct k_2-mers that survive these filters, and let $\mathtt{cluster}[1..K]$ be a vector of read identifiers that takes $K \log r$ bits of space. We also initialize a disjoint-set data structure with one cluster for each read, taking $O(r \log r)$ bits of space (see Insight 16.1).

Insight 16.1 Disjoint-set forests

Given a *fixed* set of n elements $\mathcal{A}$, assume that we want to maintain a *dynamic* collection $\mathcal{C}$ of disjoint subsets of $\mathcal{A}$ that supports the following operations:

- $\mathtt{find}(x)$: given an element $x \in \mathcal{A}$, return a unique identifier of the set in $\mathcal{C}$ that contains x;

- `union`(X, Y): given the identifiers of two sets X and Y in $\mathcal{C}$, remove X and Y from $\mathcal{C}$ and add $X \cup Y$ to $\mathcal{C}$.

A *disjoint-set forest* represents a set $X \in \mathcal{C}$ as a tree T_X, in which every node corresponds to an element of X and has a pointer to its parent in T_X. The root of T_X is used as the unique identifier of set X. To answer query `find`(x) for some $x \in X$ and $X \in \mathcal{C}$, it suffices to follow parent pointers from x to the root of T_X, and to return the root of T_X. To implement `union`(X, Y), where sets X and Y are such that $|Y| \leq |X|$, it suffices to set the parent pointer of the root of T_Y to the root of T_X: this guarantees that `find` takes $O(\log n)$ time (see Exercise 16.14).

A second optimization to speed up `find` queries consists in resetting to y the parent pointer of every node that is traversed while answering a query `find`(x), where y is the result of such query. This makes the tree flatter, reducing the time taken to answer queries about the ancestors of x and their descendants. It is possible to show that this technique achieves $O(\alpha(n))$ amortized time per operation, where $\alpha(n)$ is the inverse of the fast-growing Ackermann function $A(n, n)$. For most practical settings of n, $A(n, n)$ is less than 5.

Then, we invert BWT_R: at each step we know a position i in R, the read p this position belongs to, and the corresponding position j in BWT_R. We use the bitvector `intervals` to detect whether j belongs to the interval of a repeating k_2-mer that survived the filtering, and to get the identifier q of such a k_2-mer. If we have reached k_2-mer q for the first time, then `cluster`$[q]$ is null, and we set it to the read identifier p. Otherwise, we ask the disjoint-set data structure to merge clusters `cluster`$[q]$ and p, and we set `cluster`$[q]$ to the root of the tree created by the disjoint-set data structure after this union.

Note that the disjoint-set data structure does not need to be present in memory during this process: we can just stream to disk all queries to the disjoint-set data structure, free the memory, initialize a new disjoint-set data structure, and update it by streaming the queries back into memory. □

This algorithm can be easily adapted to cluster reads that share a k_2-mer *or its reverse complement*: see Exercise 16.9. More advanced approaches could merge two reads if they share at least two distinct k_2-mers (see Exercise 16.8), or if the regions that surround a shared k_2-mer in the two reads are within a specified edit distance. Note that the probability of erroneously merging two reads is directly proportional to the size of pre-clusters in practice, thus pre-clusters are not allowed to grow beyond a user-specified upper bound. We could then process k_2-mers *in order of increasing frequency*, stopping when the pre-clusters become too large: indeed, sharing a rare substring more likely implies that two reads come from the same genome.

Since k_2 is large, the number of occurrences of a repeating k_2-mer is approximately equal to the coverage γ of high-frequency species. Thus, the number of bits taken by the vector `cluster` is approximately $(|R|/\gamma) \log r$. This space can be further reduced by using *maximal repeats of length at least* k_2 (see Section 8.4.1 for a definition of maximal repeat). Indeed, let $\mathcal{M}_R$ denote the set of all maximal repeats of R of length at least k_2. Every repeating k_2-mer is a substring of a maximal repeat in $\mathcal{M}_R$, and distinct repeating k_2-mers might occur in R only as substrings of the same maximal repeat $W \in \mathcal{M}_R$: thus, considering just W produces an equivalent set of queries to the disjoint-set data structure. More generally, every substring V of a maximal repeat $W \in \mathcal{M}_R$ occurs in R wherever W occurs, and possibly at other positions, therefore the union operations induced by W are a subset of the union operations induced by V, and we can safely disregard W for clustering. We are thus interested in the following subset of maximal repeats of R.

DEFINITION 16.6 *Let R be a string and k be an integer. A repeat of R is called k-*submaximal *if it is a maximal repeat of R of length at least k, and if it does not contain any maximal repeat of length at least k as a substring.*

We denote by $\mathcal{M}^*_R \subseteq \mathcal{M}_R$ the set of all k_2-submaximal repeats of R. Note that $\mathcal{M}^*_R$ is at most as big as the set of distinct repeating k_2-mers of R: see Exercise 16.13.

LEMMA 16.7 *Given the bidirectional BWT index of R and an integer k_2, we can mark the intervals in $\mathsf{BWT}_{\underline{R}}$ of all k_2-submaximal repeats of R in $O(|R| \log \sigma)$ time and in $|R| + o(|R|) + \sigma \log^2 |R|$ bits of working space.*

Proof Let `intervals`$[1..|R|]$ be a bitvector in which we mark the starting position and the ending position of all intervals in $\mathsf{BWT}_{\underline{R}}$ of maximal repeats in $\mathcal{M}^*_R$. First, we mark only the intervals in $\mathsf{BWT}_{\underline{R}}$ that correspond to maximal repeats that do not contain another maximal repeat *as a suffix*. The intervals of such repeats in $\mathsf{BWT}_{\underline{R}}$ are not contained in the interval of any other maximal repeat. To do so, we traverse the suffix-link tree of R depth-first using the bidirectional BWT index of R (see Algorithm 9.3), and, as soon as we meet an internal node v of the suffix tree of R with string depth k_2, we mark in `intervals` the block associated with v in $\mathsf{BWT}_{\underline{R}}$ and we stop traversing the subtree rooted at v. We denote by $\mathcal{M}'_R$ the set of maximal repeats that results from this phase. Since $\mathcal{M}'_R$ is suffix-free, at most one repeat of $\mathcal{M}'_R$ ends at every position of R, therefore $|\mathcal{M}'_R| \leq |R|$.

In practice we may want to discard also maximal repeats that contain another maximal repeat *as a prefix*. To do so, we traverse the suffix-link tree of $\underline{R}$ in the same way as before, exchanging the role of BWT_R and $\mathsf{BWT}_{\underline{R}}$. Note that this is possible because the intervals in BWT_R and in $\mathsf{BWT}_{\underline{R}}$ are always synchronized by the bidirectional BWT index during the traversal. At the end of this process, we index `intervals` to support `rank` operations in constant time (see Section 3.2).

Then, we backward-search R *from left to right* in $\mathsf{BWT}_{\underline{R}}$, keeping the current position p in R, the corresponding position q in $\mathsf{BWT}_{\underline{R}}$, and an initially empty interval $[i..j]$. Note that this corresponds to reconstructing R *from left to right*. We use vector `intervals` to detect whether q belongs to the interval of a maximal repeat W: if so, we set $i = {}^{\bullet}\overline{W}$ and $j = \overline{W}^{\bullet}$, and we continue updating q by a backward step with character $R[p+1]$ and updating interval $[i..j]$ by backward-searching string $W \cdot R[p+1]$. Assume that, at some point, q belongs again to the interval of a maximal repeat V. If ${}^{\bullet}\overline{V}^{\bullet} \subseteq [i..j]$, then the occurrence of V that ends at position p in R includes the previous occurrence of W, therefore W is a substring of V and the interval of V can be unmarked from `intervals`. Otherwise, if $[i..j] \subset \overleftrightarrow{V}$, then the occurrence of V that ends at position p in R cannot include the previous occurrence of W: we thus reset $[i..j]$ to $\overleftrightarrow{V}$, and continue. □

We can then use the bitvector `intervals` to cluster reads as described in Lemma 16.5. Other approaches build pre-clusters using long suffix–prefix overlaps (see Section 8.4.4), or long maximal exact matches (see Section 11.1.3): note that both of these structures are maximal repeats of R.

16.2.3 Growing clusters

As mentioned, the pre-clusters created in Section 16.2.2 typically correspond to short, possibly noncontiguous substrings of a single genome: we thus need to merge pre-clusters that belong to the same genome. Before proceeding, we can clearly filter out pre-clusters with too few reads, or such that the string which would result from assembling the reads in the pre-cluster would be too short: the reads in such pre-clusters can be processed in a separate phase. If we have paired-end information, we can also merge two pre-clusters if they contain corresponding paired reads. In general, we want to merge two pre-clusters if they have similar k_4-mer composition, where $k_4 < k_2$ is a user-specified constant estimated from known genomes.

Owing to the existence of repeating k_2-mers inside the same pre-cluster, we estimate the k_4-mer composition not directly from the reads in a pre-cluster, but from long, repeating substrings in the pre-cluster. Specifically, we extract all the maximal repeats of length at least $k_3 > k_2$ that occur at least τ_3 times in a pre-cluster. This operation could be carried out for each pre-cluster separately. However, since $k_3 > k_2$, a maximal repeat of length at least k_3 occurs in exactly one pre-cluster in R: we can thus compute all the maximal repeats of length at least k_3 *in the whole of* R by enumerating the internal nodes of the suffix tree of R just once (see Section 11.1.1), we could mark their intervals in BWT_R, and then we could extract the corresponding strings by inverting BWT_R: the details are left to Exercise 16.15. The parameters k_3 and τ_3 are again computed from estimates of the error rate and coverage, to minimize the probability that a k_3-mer in a genome occurs fewer than τ_3 times in a read set due to sequencing errors, and to minimize the probability that spurious k_3-mers are generated in a read set by sequencing errors.

Finally, we compute the k_4-mer composition of all the maximal repeats of a pre-cluster, and we input all such composition vectors to any clustering algorithm, for example k-means (see Insight 16.2). At the end of this step, all reads sampled from high-frequency species have been clustered. Reads from low-frequency species can be processed with a similar pipeline (see Figure 16.1): first, we filter out all reads that contain only *unique* k_1*-mers*; that is, we lower to two the filtering threshold τ_1 used in Section 16.2.1. Such reads either have a very high number of errors, or they belong to extremely low-frequency species for which there is too little information in the dataset to perform a reliable clustering. We then pre-cluster the remaining reads, using either

k_2'-mers or maximal repeats of length at least k_2', where $k_2' < k_2$. In practice we might want to organize this step in multiple rounds, in which we use decreasing values of k_2'. The k_4'-mer composition of the resulting pre-clusters (with $k_4' < k_4$) is then used to build new clusters.

Insight 16.2 *k*-means clustering

Let $\mathcal{S} \subset \mathbb{R}^n$ be a set of vectors, and let $\mathcal{S}^1, \mathcal{S}^2, \ldots, \mathcal{S}^k$ be a partitioning of $\mathcal{S}$ into k disjoint subsets, called *clusters*. Consider the cost function $\sum_{i=1}^{k} \sum_{\mathbf{X} \in \mathcal{S}^i} ||\mathbf{X} - \mathbf{M}^i||^2$, called the *within-cluster sum of squares*, where $\mathbf{M}^i$ is the mean of all vectors in cluster $\mathcal{S}^i$, and $|| \cdot ||$ is the 2-norm described in Section 11.2. It can be shown that the problem of finding the set of k clusters $\mathcal{S}^1, \mathcal{S}^2, \ldots, \mathcal{S}^k$ that minimize the within-cluster sum of squares is NP-hard. However, given an initial set $\mathbf{M}^1, \mathbf{M}^2, \ldots, \mathbf{M}^k$ of k mean vectors, we could find a local minimum of the cost function by iteratively assigning each vector $\mathbf{X} \in \mathcal{S}$ to a mean vector $\mathbf{M}^i$ that minimizes $|\mathbf{X} - \mathbf{M}^i|^2$, and updating every mean vector to the average of all vectors assigned to it in the previous iteration. This iterative approach is called *k-means clustering*.

Assume that each vector $\mathbf{X} \in \mathcal{S}$ is the composition vector of a corresponding string $X \in [1..\sigma]^+$, with one component for every distinct substring of a given length ℓ. In other words, $\mathbf{X}[W]$ is the frequency of ℓ-mer W in X, divided by the length of X (see Section 11.2.1). Assume also that we have the bidirectional BWT index of every string X in the set. We can perform k-means clustering *without building the composition vectors* $\mathbf{X} \in \mathcal{S}$ *explicitly*, as follows.

We use the bidirectional BWT index of X to mark in a bitvector $\mathtt{intervals}_X[1..|X|]$ the first position of the interval in $\mathsf{BWT}_{X\#}$ of every ℓ-mer that occurs in X, as described in Section 9.7.3. Then, we index $\mathtt{intervals}_X$ to support rank queries in constant time, and we initialize another bitvector, $\mathtt{used}_X[1..L_X]$, where L_X is the number of distinct ℓ-mers in X. Computing the distance between $\mathbf{X}$ and $\mathbf{M}^i$ amounts to inverting $\mathsf{BWT}_{X\#}$: the first time we reach the interval of an ℓ-mer W of X, we compute $|\mathbf{X}[W] - \mathbf{M}^i[W]|^2$ and we flag the interval of W in the bitvector $\mathtt{used}$. The contribution of the ℓ-mers with a non-zero component in $\mathbf{M}^i$ and a zero component in $\mathbf{X}$ can be computed using a bitvector that flags the non-zero components of $\mathbf{M}^i$ that have been found in X. Updating $\mathbf{M}^i$ also amounts to inverting $\mathsf{BWT}_{X\#}$ for each X that has been assigned to $\mathbf{M}^i$ in the previous phase: the first time we reach the interval of an ℓ-mer W in $\mathsf{BWT}_{X\#}$, we add to its component in a new vector $\mathbf{M}^i$ (initially set to zero) the size of the interval divided by the length of X, and we flag the interval of W in the bitvector $\mathtt{used}$.

16.3 Comparing metagenomic samples

Assume that we have a database $\mathcal{D}$ of known metagenomes. Given a new metagenome $\mathcal{R}$, we want to determine which metagenomes in $\mathcal{D}$ are most similar to $\mathcal{R}$. One way of doing this could be to use the algorithms described in Section 16.1 to estimate the set of taxa $\mathcal{S}$ present in $\mathcal{R}$ and their relative frequencies $\mathbf{P}$: assuming that all metagenomes in $\mathcal{D}$ have been annotated in this way, we could use such taxonomic compositions to retrieve similar metagenomes.

16.3.1 Sequence-based methods

When the species in $\mathcal{R}$ and in $\mathcal{D}$ are mostly unknown, we need to compare metagenomes using their sequence composition rather than their taxonomic composition. One way of doing this is by using the k-mer and substring kernels described in Section 11.2.

Note that the relative frequency of a k-mer W in $\mathcal{R}$ is affected both by the frequency of W in the distinct genomes of the sample and by the relative frequencies $\mathbf{P}$ of such genomes. Therefore, it could happen that samples containing very different sets of *distinct genomes* display a very similar k-mer composition, because of the relative frequencies of the genomes in the samples. We might thus be interested in estimating the k-mer composition vectors *of the distinct genomes in a metagenome*, for example by preliminarily clustering or assembling the reads as described in Section 16.2 and Chapter 13.

16.3.2 Read-based methods

Rather than comparing two metagenomic samples at the substring level, we could compare them *at the read level*, by estimating the proportion of reads they have in common.

DEFINITION 16.8 *Let $\mathcal{R} = \{R^1, R^2, \ldots, R^r\}$ and $\mathcal{Q} = \{Q^1, Q^2, \ldots, Q^q\}$ be two metagenomic samples. We say that a read $R^i \in \mathcal{R}$ is* similar *to a read $Q^j \in \mathcal{Q}$ if there are at least τ, not necessarily distinct, substrings of length k that occur in R^i and in Q^j in the same order and without overlaps, where $k < \ell$ and $\tau \geq 1$ are two user-specified parameters.*

Note that the same read R^i can be similar to multiple reads in $\mathcal{Q}$. Denote by $\mathcal{R} \otimes \mathcal{Q}$ the set of reads in $\mathcal{R}$ that are similar to at least one read in $\mathcal{Q}$, and denote by $\mathcal{Q} \otimes \mathcal{R}$ the set of reads in $\mathcal{Q}$ that are similar to at least one read in $\mathcal{R}$. We could estimate the similarity between $\mathcal{R}$ and $\mathcal{Q}$ by computing $f(\mathcal{R}, \mathcal{Q}) = (|(\mathcal{R} \otimes \mathcal{Q})| + |(\mathcal{Q} \otimes \mathcal{R})|)/(|\mathcal{R}| + |\mathcal{Q}|)$, or equivalently the proportion of reads that are similar to another read in the union of the two read sets. This can be done by adapting the approach of Exercise 16.8.

In order to scale to larger datasets, we might want to waive the requirement of computing $\mathcal{R} \otimes \mathcal{Q}$ and $\mathcal{Q} \otimes \mathcal{R}$ exactly. For example, we could approximate $\mathcal{R} \otimes \mathcal{Q}$ with the set $\mathcal{R} \odot \mathcal{Q}$ that contains all reads $R^i \in \mathcal{R}$ with at least τ nonoverlapping substrings of length k that occur also in $\mathcal{Q}$, but not necessarily in the same read Q^i and not necessarily in the same order. Setting k long enough makes $\mathcal{R} \odot \mathcal{Q}$ sufficiently close to $\mathcal{R} \otimes \mathcal{Q}$ in practice. One way of computing $\mathcal{R} \odot \mathcal{Q}$ consists in building the matching statistics vector MS_{R^i} of each read $R^i \in \mathcal{R}$ with respect to the string that results from concatenating all reads in $\mathcal{Q}$, as described in Section 11.2.3. Then, we can create a directed acyclic graph $G = (V, E)$ for each read R^i, whose vertices correspond to all positions $p \in [1..\ell]$ with $\mathsf{MS}_{R^i}[p] \geq k$, and whose arcs $(v_p, v_q) \in E$ correspond to pairs of positions $p < q$ such that $q - p \geq k$. A longest path in G is a set of nonoverlapping substrings of length k of maximum size, and the length of such a path can be computed in $O(|V| + |E|)$ time by dynamic programming: see Exercise 16.16.

16.3.3 Multi-sample methods

Assume that we have multiple metagenomic samples of type 1 and multiple metagenomic samples of type 2: a substring that is common to all samples of type 1 and that does not occur in any sample of type 2 could be used to discriminate between type 1 and type 2, and possibly to build a model of the two classes. Recall the *document counting problem* described on page 149 (Problem 8.4): given a set of d strings, we want to report, for every maximal repeat in the set, the number of strings that include the repeat as a substring. One can extend this problem and its linear-time solution described in Theorem 8.21 to consider two sets, one containing type 1 metagenomic samples and the other containing type 2 metagenomic samples, and to report, for every maximal repeat in the union of the two sets, the number of times it occurs in type 1 samples and in type 2 samples as a substring. Note that this method is not restricted to a fixed length k, and it can be made scalable since document counting can be solved space-efficiently: see Exercise 11.3 on page 257.

16.4 Literature

The species estimation strategy that combines read mapping with a lowest common ancestor is described in Huson *et al.* (2011) and references therein, and some of its variants using contigs and read clusters are detailed in Wang *et al.* (2014a). The idea of re-mapping the best hit of a read to the set of all its hits is from Haque *et al.* (2009). The variant of lowest common ancestor based on the set cover problem is described in Gori *et al.* (2011). Further heuristics based on detecting open reading frames in reads are detailed in Sharma *et al.* (2012). The idea of using taxonomic markers to speed up frequency estimation is described in Liu *et al.* (2011) and Segata *et al.* (2012), where markers are entire genes rather than arbitrary substrings, and in Edwards *et al.* (2012), where k-mers are used to mark protein families. The idea of computing markers from matching statistics and from the BWT of genomes was developed for this book. Markers are related to the notion of *cores* (substrings that are present in all genomes of a taxon v) and of *crowns* (cores of a taxon v that are not cores of $\mathtt{parent}(v)$, or equivalently strings that can separate v from its siblings in a taxonomy): see Huang *et al*. (2013).

Problem 16.1 is adapted from Lo *et al.* (2013) and its generalization to Problem 16.2 is from Sobih *et al.* (2015). The NP-hardness proof for Problem 16.2 is from R. Rizzi (personal communication, 2014). This problem formulation has been applied to species estimation in Sobih *et al.* (2015). A similar objective of obtaining a predefined read coverage, expressed again with the formalism of minimum-cost flows, appears in Medvedev *et al.* (2010) in connection with a copy-number variation problem.

The read clustering pipeline in Section 16.2 is from Wang *et al.* (2012a): this paper contains additional details on estimating the probability of merging two reads from different genomes used in Section 16.2.2, and on the expected error for a specific setting of parameters k_3 and τ_3 used in Section 16.2.3. The space-efficient implementation of the pipeline and the notion of k-submaximal repeats were developed for this book, and are detailed in Alanko *et al.* (2014). The idea of performing k-means clustering without storing the composition vectors of the input strings explicitly was developed for this book. Filtering out low-frequency species to improve the clustering accuracy has been proposed in Wang *et al.* (2012b). An alternative clustering criterion based on rare or unique k-mers appears in Haubold *et al.* (2005) and Tanaseichuk *et al.* (2011). Using maximal repeats to cluster reads has been described in Baran & Halperin (2012); conversely, using contigs produced by an assembler has been described in Alneberg *et al.* (2013) and Wang *et al.* (2014a). More details on the approach of combining multiple metagenomic samples that contain the same species before clustering appear in Baran & Halperin (2012). Alneberg *et al.* (2013) use the coverage of a contig or of a cluster in each sample of a collection, in addition to its k-mer composition, to improve clustering accuracy.

A method for comparing metagenomic samples using their estimated taxonomic composition, and its efficient implementation on GPU, is described in Su *et al.* (2013). More information about the k-mer composition of metagenomic samples can be found in Willner *et al.* (2009), and studies on the performance of a number of similarity measures based on taxonomic and k-mer composition can be found in Su *et al.* (2012), Jiang *et al.* (2012), and Wang *et al.* (2014b). The idea of comparing metagenomes at the read level using operator $\odot$ is from Maillet *et al.* (2012). This paper solves Exercise 16.16.

The space-efficient approach for comparing multiple metagenomic samples using document counting is from Fischer *et al.* (2008). As multi-samples become larger, even such a space-efficient solution might not be scalable in practice, unless executed on a machine with several terabytes of main memory. The distributed algorithm described in Välimäki & Puglisi (2012) scales this approach to a cluster of standard computers: this solution was applied to metagenomic samples in Seth *et al.* (2014).

Exercises

16.1 Show that Problem 16.1 can be reduced to a minimum-cost network flow problem, by extending the reduction constructed in Section 5.3.3 for Problem 5.7.

16.2 Show that you can reduce Problem 16.1 to one in which every read vertex in A has at least two incident edges.

16.3 Discuss how to modify the statement of Problem 16.1 if the probability of sequencing a read from a certain location inside each genome is no longer uniform for all locations, but follows a different distribution function, which is known beforehand.

16.4 Explain what changes you need to make to the reduction constructed in Exercise 16.1 if the objective function in Problem 16.1 is

$$(1-\alpha)\sum_{(x,y)\in M} c(x,y) + \alpha \sum_{i=1}^{m}\sum_{j=1}^{s_i}\left(\frac{c_i}{s_i} - d_M(y_{i,j})\right)^2 . \tag{16.3}$$

Hint. Recall Exercise 5.16.

16.5 What can you say about the complexity of Problem 16.2 for $\alpha = 0$? What about $\alpha = 1$?

16.6 Describe an algorithm to compute the BWT of the reads that are kept and of the reads that are filtered out in Section 16.2.1, by reusing the BWT of the original file. Recall that reads in the input file R are separated by distinct binary strings on the artificial characters $\#_0$ and $\#_1$ which do not belong to $[1..\sigma]$.

16.7 Given the concatenation R of all reads in a metagenomic sample as defined in Section 16.2, consider the interval of a k_2-mer W in BWT_R. Section 16.2.2 claims that, during the creation of pre-clusters, we can discard any k_2-mer V whose interval in BWT_R coincides with the interval of aW for some $a \in [1..\sigma]$. Prove this claim, and describe an algorithm that implements it in linear time using just one scan of the bitvector `intervals`.

16.8 With reference to Section 16.2.2, describe an algorithm to create pre-clusters in which two reads are merged if they share *at least two* distinct k_2-mers. Assume that you have $|R| \log r$ bits of main memory in addition to the space required by Lemma 16.5, and assume that you have an algorithm that sorts tuples in external memory as a black box. Describe how to implement additional conditions on top of your algorithm, for example that the two shared k_2-mers are in the same order in both reads, that they do not overlap, or that they are offset by exactly one position.

16.9 With reference to Section 16.2.2, describe an algorithm to create pre-clusters in which two reads are merged if they share a k_2-mer *or its reverse complement*. Note that in this case we cannot discard k_2-mers that occur exactly once in R, since they could occur twice in $R \cdot \underline{R}$. Describe a solution that takes $K + K' \log r$ bits of space in addition to the bidirectional BWT index of R, where K is the number of distinct k_2-mers in R, and K' is the number of distinct k_2-mers that occur at least twice in $R \cdot \underline{R}$. This algorithm should perform $k_2 + 1$ operations for every position of the input file. Adapt your solution to filter out reads such that all their k_1-mers occur less than τ_1 times in $R \cdot \underline{R}$, as described in Section 16.2.1. Finally, describe another solution for creating pre-clusters that uses string $R \cdot \underline{R}$ as input and that takes approximately twice the time and the space of Lemma 16.5.

16.10 Describe a way to parallelize the algorithm in Lemma 16.5 by using samples of SA_R (see Section 9.2.3).

16.11 Adapt the algorithm for document counting described in Section 8.4.2 such that, given a reference taxonomy $\mathcal{T}$, it computes the markers of every taxon $v \in \mathcal{T}$.

16.12 Let $\mathsf{MS}_{S,T,\tau}$ be the following generalization of the matching statistics vector described in Section 11.2.3: $\mathsf{MS}_{S,T,\tau}[i]$ is the longest prefix of suffix $S[i..|S|]\#$ that occurs at least τ times in T. Similarly, let $\mathsf{SUS}_{T,\tau}[i]$ be the shortest prefix of $T[i..|T|]\#$ that occurs at most τ times in T. Describe an algorithm that computes the markers of every taxon v in a reference taxonomy $\mathcal{T}$ using the shortest unique substring vector $\mathsf{SUS}_{S,\tau}$, the matching statistics vector $\mathsf{MS}_{S,T,\tau+1}$, and a bitvector $\mathtt{flag}_{S,T,\tau}[1..|S|]$ such that $\mathtt{flag}_{S,T,\tau}[i] = 1$ if and only if $S[i..i + \mathsf{MS}_{S,T,1}[i] - 1]$ occurs at most τ times in T, for every genome $T \neq S$ in the database.

16.13 Prove that the number of k-submaximal repeats of a string S is upper-bounded by the number of distinct k-mers that occur at least twice in S.

16.14 With reference to Insight 16.1, prove that attaching the root of the smallest subtree to the root of the largest subtree guarantees $O(\log n)$ time for `find`.

16.15 With reference to Section 16.2.3, give the pseudocode of an algorithm that extracts all the maximal repeats of length at least k_4 from all pre-clusters, using a single enumeration of all the internal nodes of the suffix tree of R.

16.16 With reference to Section 16.3.2, describe a way to approximate $\mathcal{R} \otimes \mathcal{Q}$ that is more accurate than just computing $\mathcal{R} \odot \mathcal{Q}$. *Hint.* Use the operator $\odot$ iteratively on suitable subsets of $\mathcal{R}$ and $\mathcal{Q}$. Then, give the pseudocode of an algorithm that, given vector MS_{R^i} for a read R^i, finds a set of nonoverlapping substrings of length k of maximum size.