13 Fragment assembly

In the preceding chapters we assumed the genome sequence under study to be known. Now it is time to look at strategies for how to *assemble* fragments of DNA into longer contiguous blocks, and eventually into chromosomes. This chapter is partitioned into sections roughly following a plausible workflow of a de novo assembly project, namely, *error correction*, *contig assembly*, *scaffolding*, and *gap filling*. To understand the reason for splitting the problem into these realistic subproblems, we first consider the hypothetical scenario of having error-free data from a DNA fragment.

13.1 Sequencing by hybridization

Assume we have separated a single DNA strand spelling a sequence T, and managed to measure its *k-mer spectrum*; that is, for each k-mer W of T we have the frequency $\texttt{freq}(W)$ telling us how many times it appears in T. *Microarrays* are a technology that provides such information. They contain a slot for each k-mer W such that fragments containing W hybridize to the several copies of the complement fragment contained in that slot. The amount of hybridization can be converted to an estimate on the frequency count $\texttt{freq}(W)$ for each k-mer W. The *sequencing by hybridization* problem asks us to reconstruct T from this estimated k-mer spectrum.

Another way to estimate the k-mer spectrum is to use high-throughput sequencing on T; the k-mer spectrum of the reads normalized by average coverage gives such an estimate.

Now, assume that we have a perfect k-mer spectrum containing no errors. It turns out that one can find in linear time a sequence T' having exactly that k-mer spectrum. If this problem has a unique solution, then of course $T' = T$.

The algorithm works by solving the Eulerian path problem (see Section 4.2.1) on a de Bruijn graph (see Section 9.7) representing the k-mers.

Here we need the following *expanded* de Bruijn graph $G = (V, E)$. For each k-mer $W = w_1 w_2 \cdots w_k$ we add $\texttt{freq}(W)$ copies of the arcs (u, v) to E such that vertex $u \in V$ is labeled with the $(k-1)$-mer $\ell(u) = w_1 w_2 \cdots w_{k-1}$ and vertex $v \in V$ is labeled with the $(k-1)$-mer $\ell(v) = w_2 w_3 \cdots w_k$. Naturally, the arc (u, v) is then labeled $\ell(u, v) = W$. The set V of vertices then consists of these $(k-1)$-mer prefixes and suffixes of the k-mers, such that no two vertices have the same label. That is, the arcs form a multiset,

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

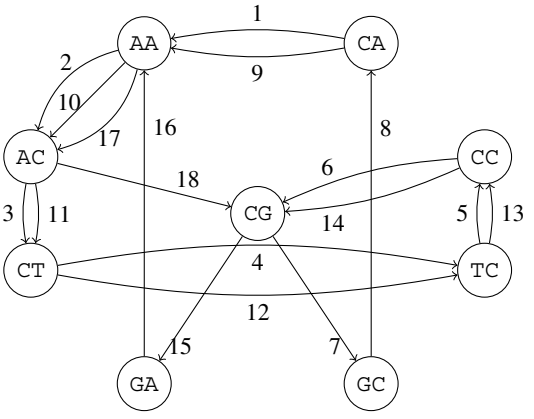


Figure 13.1 The expanded de Bruijn graph of the k-mer spectrum $\texttt{freq(AAC)} = 3$, $\texttt{freq(ACG)} = 1$, $\texttt{freq(ACT)} = 2$, $\texttt{freq(CAA)} = 2$, $\texttt{freq(CCG)} = 2$, $\texttt{freq(CGA)} = 1$, $\texttt{freq(CGC)} = 1$, $\texttt{freq(CTC)} = 2$, $\texttt{freq(GAA)} = 1$, $\texttt{freq(GCA)} = 1$, and $\texttt{freq(TCC)} = 3$. The numbers on the arcs denote the order in which they appear in an Eulerian path spelling $T' = \texttt{CAACTCCGCAACTCCGAACG}$.

and the vertices form a set. Figure 13.1 illustrates the construction of this expanded de Bruijn graph.

An Eulerian path $v_1, v_2, \ldots, v_{n-k+1}$ in the expanded de Bruijn graph G visits all arcs, and it can be interpreted as a sequence $T' = \ell(v_1) \cdot \ell(v_2)[k-1] \cdot \ell(v_3)[k-1] \cdot \cdots \cdot \ell(v_{n-k+1})[k-1]$. Each arc of G has its label associated with a unique position in T'. Hence, T' has the same k-mer spectrum as the one from which the expanded de Bruijn graph was built. As shown in Section 4.2.1, this path can be found in linear time. One such path is shown in Figure 13.1.

Obviously, one issue with the above approach is that the k-mer spectrum is extremely hard to estimate perfectly. Having an inaccurate spectrum, one could formulate a problem asking how to correct the graph minimally so that it contains an Eulerian path. However, this formulation makes the problem hard (see the literature section).

Another possibility is to formulate a problem where one is given for every k-mer (that is, an arc of the expanded de Bruijn graph) a lower bound on the number of times it must appear in the reconstructed string T. Exercise 13.1 asks the reader to show that this problem, called the Chinese postman problem, can be reduced to a network flow problem (recall Chapter 5).

Another issue is that there can be several sequences having the same k-mer spectrum. Moreover, if one estimates the spectrum with high-throughput sequencing, one should also take into account the facts that reads come from both DNA strands and from both haplotypes of a diploid individual.

We shall next explore practical assembly strategies that take into account the above realistic constraints. We will come back to a combinatorial modeling, and give a proof of the computational hardness of the assembly problem when studying *scaffolding*, since this subproblem best captures the nature of the entire fragment assembly problem.

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

13.2 Contig assembly

Instead of trying to estimate an entire sequence using a k-mer spectrum, a de Bruijn graph, or read overlaps, the *contig assembly* problem has a more modest goal: find a set of paths in a given *assembly graph* such that the contiguous sequence content (called *contig*) induced by each path is likely to be present in the genome under study. Naturally, the goal is to find as few and as long contigs as possible, without sacrificing the accuracy of the assembly.

Here an assembly graph refers to any directed graph whose vertices represent plausible substrings of the genome and whose arcs indicate overlapping substrings. The de Bruijn graph constructed from the k-mers of the sequencing reads is one example of an assembly graph. Another example is an *overlap graph* based on maximal exact/approximate overlaps of reads, whose construction we covered in Section 10.4. A characteristic feature of an assembly graph is that a path from a vertex s to a vertex t can be interpreted as a plausible substring of the genome, obtained by merging the substrings represented by the vertices on the path. In the previous section, we saw how this procedure was carried out for de Bruijn graphs, and we will later detail this procedure on other kinds of assembly graphs.

The simplest set of contigs to deduce from an assembly graph is the one induced by *maximal unary paths*, also known as *unitigs* for *unambiguous contigs*.

DEFINITION 13.1 *A path $P = (s, v_1, \ldots, v_n, t)$ from s to t in a directed graph G is called* maximal unary *if all of the v_i have exactly one in-neighbor and one out-neighbor (on P), and s and t do not have this property.*

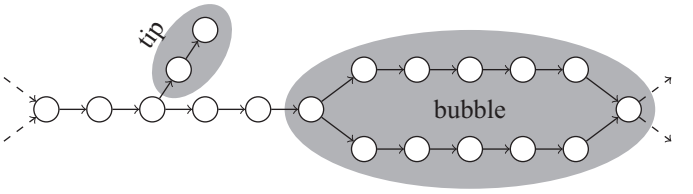
Unfortunately, the contigs extracted from maximal unary paths from an assembly graph built on a set of *raw* reads will be too short and redundant due to substructures caused by sequencing errors and alleles. *Cleaning* an assembly graph is discussed in Insight 13.1. Exercise 13.22 asks the reader under which assumptions a unary path in an assembly graph correctly spells a contig. Exercise 13.23 asks for an algorithm to extract all maximal unary paths in linear time by a simple graph traversal. Exercise 13.24 asks for an alternative algorithm based on a compaction process of the directed graph. Finally, Exercise 13.25 further formalizes the notion of a contig.

Insight 13.1 Tips, bubbles, and spurious arcs

Consider a de Bruijn graph built on the k-mers of a set of reads that may contain sequencing errors. A sequencing error may cause an otherwise unary path to branch at one point with some short unary path that ends in a sink (or, symmetrically, a short unary path starting from a source ending inside an otherwise unary path). Such substructures are called *tips*; these are easy to remove, given a threshold on the maximum length of such paths. Sequencing error in the middle of a read may also cause a *bubble*, in which two vertices are connected by two parallel unary paths. Contig extraction through maximal unary paths would report both parallel

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

paths of a bubble, although they might represent the same area in the genome. These substructures are visualized below:



In addition to *erroneous bubbles* caused by sequencing errors, there can be *redundant bubbles*. These are caused by alleles where reads (or k-mers) containing the same allele type form one path of a bubble. In diploid organisms, one should then expect to see a bubble in the de Bruijn graph for each heterozygous mutation. This gives in fact a way to do *de novo variation calling* without a reference genome, as opposed to the alignment-based variation calling that we will cover in Section 14.1.

Finally, bubbles can also appear accidentally when source and sink k-mers appear at many places in the genome. Such bubbles are likely to have different sequence content in the two parallel paths, and hence one can try to separate them from erroneous and redundant bubbles.

For fragment assembly, it is useful to clean the graph of erroneous and redundant bubbles, by removing one of their parallel paths. Since we can label each arc by the amount of reads containing that k-mer, it is easy to select the lower-coverage path for removal. This avoids some redundancy in the resulting contigs. After the assembly has been finished, the variants can be reconstructed, for example by read alignment (see Section 14.1).

Sequencing errors can also create more complex substructures that are difficult to detect. A possible strategy to clean the graph more aggressively is to remove *spurious arcs*, that is, arcs with low support compared with their neighbors.

13.2.1 Read error correction

A practical problem in cleaning an assembly graph, as discussed in Insight 13.1, is that one should modify the assembly graph dynamically. The succinct structures described in Section 9.7 are, however, static. A workaround is to do *read error correction/removal* based on the identified substructures of the assembly graph and build the assembly graph again using the corrected reads. The advantage is that one can do contig assembly using a different assembly graph than the one used for correcting the reads.

To fix ideas, consider that we have identified tips, bubbles, and spurious arcs in a de Bruijn graph, as defined in Insight 13.1. We wish to correct the reads so that the identified tips and spurious arcs do not appear in the de Bruijn graph built on the corrected reads. Moreover, each erroneous or redundant bubble should be replaced by one of its two unary paths, selected by some criterion, as discussed in Insight 13.1.

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

A tip spells a string that must overlap either prefixes of some reads or suffixes of some reads by at least k characters. In Section 8.4.4, we studied how to find such exact overlaps, and in Section 13.2.3 we will study a space-efficient algorithm for this problem. That is, we can build a BWT index on the set of reads, occupying $O(N \log \sigma)$ bits, where N is the total length of the reads, so that each read whose prefix matches the suffix of a tip by at least k characters will be reported. It is then easy to cut the corresponding overlap from the reported reads. Building the BWT index on the reversed reads and repeating the procedure on reversed tips solves the symmetric case.

A bubble b spells two strings, say A^b and B^b, where A^b has been identified as erroneous or redundant and B^b as the consensus. That is, we have a set of replacement rules $S = \{(A^b \rightarrow B^b) \ : \ b \text{ is a bubble}\}$. A BWT index on the set of reads can again be used to implement these replacement rules: search each A^b, locate all its occurrences, and replace substring A^b with B^b inside each read found.

A spurious arc is a locally under-represented k-mer, meaning that the source of the corresponding arc has at least one other outgoing arc with significantly more support. Hence, one can consider a replacement rule like in the case of bubbles.

Exercise 13.3 asks you to consider how to implement the replacement rules in detail. Exercise 13.4 asks you to develop a linear-time algorithm to detect all bubbles in an assembly graph.

13.2.2 Reverse complements

One aspect we have omitted so far is that reads can come from both strands of a DNA fragment. This is addressed in Insight 13.2 for the case of overlap graphs. We will later learn how to structurally compress overlap graphs, and such compaction is also defined on the special structure designed in Insight 13.2.

With de Bruijn graphs, one could use the same reduction as in the case of overlap graphs. We will not consider this variant in detail, since the succinct representations of Section 9.7 do not easily extend to this special de Bruijn graph structure. A trivial solution is, however, to build the de Bruijn graph on the set of reads and their reverse complements. This will result in redundant contigs representing opposite strands that need to be detected afterwards.

In the next subsections, we assume for simplicity that reads come from one strand, but one can modify those approaches similarly to what is done in Insight 13.2 to cope with the real setting. Later, in the scaffolding step we need to explicitly consider reverse complementation again.

Insight 13.2 Reverse complements in assembly graphs

Let us consider an overlap graph, whose vertices represent reads, and whose arcs represent their overlaps. Each read can originate from either one of the two DNA strands, and recall that the strands are sequenced in opposite directions. Let us consider all eight ways in which two reads R and S can overlap (denote by $\underline{R}$ and $\underline{S}$ the reverse complements of R and S, respectively):

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

(i) a suffix of R overlaps a prefix of S;
(i′) a suffix of $\underline{S}$ overlaps a prefix of $\underline{R}$;
(ii) a suffix of R overlaps a prefix of $\underline{S}$;
(ii′) a suffix of S overlaps a prefix of $\underline{R}$;
(iii) a suffix of $\underline{R}$ overlaps a prefix of S;
(iii′) a suffix of $\underline{S}$ overlaps a prefix of R;
(iv) a suffix of $\underline{R}$ overlaps a prefix of $\underline{S}$;
(iv′) a suffix of S overlaps a prefix of R.

Observe that, if two reads overlap as in case (i) on one strand, then on the other strand we have an overlap as in (i′). Symmetrically, if two reads overlap as in (i′) on one strand, then on the other strand we have the overlap (i). A similar analogy holds for the overlaps (ii)–(ii′), (iii)–(iii′), and (iv)–(iv′).

A trivial solution for dealing with these eight cases is to add, for each read, two separate vertices: one for the read as given and one for its reverse complement. However, this has the problem that both of these vertices can appear in some assembled fragment, meaning that the read is interpreted in two ways.

Another solution is to use a modified graph structure that has some added semantics. Split the vertex corresponding to each read R into two vertices $R_{\texttt{start}}$ and $R_{\texttt{end}}$ and connect these vertices with the undirected edge $(R_{\texttt{start}}, R_{\texttt{end}})$, which we call the *read edge*. If a path reaches $R_{\texttt{start}}$ and then continues with $R_{\texttt{end}}$, then this path spells the read R as given; if it first reaches $R_{\texttt{end}}$ and then continues with $R_{\texttt{start}}$, then this path spells the reverse complement of R.

The eight overlap cases now correspond to the following *undirected* edges:

i. $(R_{\texttt{end}}, S_{\texttt{start}})$ for an overlap as in the above cases (i) or (i′);
ii. $(R_{\texttt{end}}, S_{\texttt{end}})$ for an overlap as in the above cases (ii) or (ii′);
iii. $(R_{\texttt{start}}, S_{\texttt{start}})$ for an overlap as in the above cases (iii) or (iii′);
iv. $(R_{\texttt{start}}, S_{\texttt{end}})$ for an overlap as in the above cases (iv) or (iv′).

A path in this mixture graph must start and end with a read arc, and read arcs and overlap arcs must alternate. The direction of the read arcs visited by a path determines whether a read is interpreted as given or as reverse complemented.

13.2.3 Irreducible overlap graphs

Succinct representations for de Bruijn graph were studied in Section 9.7. They enable space-efficient contig assembly. However, overlap graphs contain more contiguous information about reads, possibly resulting in more accurate contigs. So it makes sense to ascertain whether they also could be represented in small space in order to obtain a scalable and accurate contig assembly method. Some general graph compression data structures could be applied on overlap graphs, but it turns out that this does not significantly save space. We opt for *structural compression*; some overlaps are redundant and can be removed. This approach will be developed in the following.

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

Recall that we studied in Section 8.4.4 how to efficiently find maximal exact overlaps between reads using the suffix tree. A more space-efficient approach using the BWT index was given to find approximate overlaps in Section 10.4. We can hence build an overlap graph with reads as vertices and their (approximate) overlaps as arcs. For large genomes, such an approach is infeasible due to the size of the resulting graph. Now that we have studied how to correct sequencing errors, we could resort back to an overlap graph defined on exact overlaps. We will next study how to build such a graph on maximal exact overlaps with the suffix tree replaced (mostly) by the BWT index, to achieve a practical space-efficient algorithm. Then we proceed with a practical algorithm to build directly an *irreducible overlap graph*, omitting arcs that can be derived through an intermediate vertex.

Space-efficient computation of maximal overlaps

Consider a set of reads $\mathcal{R} = \{R^1, R^2, \ldots, R^d\}$. We form a concatenation $C = \#R^1\#R^2\#\cdots\#R^d\#$ and build a BWT index on it. Note that every position i of an occurrence of # can easily be mapped to the lexicographic order of the corresponding read through the operation $\texttt{rank}_{\#}(\mathsf{BWT}_C, i) - 1$. We also store the array $\texttt{RA}[1..d] = \texttt{SA}[2..d+1]$, which gives the starting positions of the reads in the concatenation, given their lexicographic order. In some cases, it might be more convenient that the reads are stored in lexicographic order, so that the lexicographic number returned by the BWT index represents the sequential order in which the read is stored. We can put the reads into lexicographic order, by scanning the array RA, to retrieve the starting position of the reads in lexicographic order and concatenate them together. We then rebuild the BWT index and the array RA. Exercise 13.10 asks you for an alternative way to sort the reads in $O(N \log \sigma)$ time and bits of space, where N is the length of the reads concatenation.

Consider a backward search with read R^i on the BWT index of C. We obtain an interval $[p_k..q_k]$ in the BWT of C associated with each suffix of $R^i_{k..|R^i|}$. If a backward step with # from $[p_k..q_k]$ results in a non-empty interval $[p_k^{\#}..q_k^{\#}]$, there is at least one suffix of C in the interval $[p_k..q_k]$ of the suffix array of C that starts with an entire read R^j.

Given a threshold τ on the minimum overlap, we should consider only intervals $[p_k^{\#}..q_k^{\#}]$ with $k \leq |R^i| - \tau$. One could list all pairs (i, j) as arcs of an overlap graph with $j \in [p_k^{\#} - 1..q_k^{\#} - 1]$. However, there are cases where the same (i, j) can be reported multiple times; this problem was solved using a suffix tree in Section 8.4.4, but now we aim for a more space-efficient solution. We exploit the following observation on the relationship between the problematic intervals.

LEMMA 13.2 *If reads R^i and R^j have two or more suffix–prefix overlaps, the shorter overlapping strings are prefixes of the longer overlapping strings.*

Proof It is sufficient to consider two overlaps of length o_1 and o_2 with $o_1 < o_2$. We have that $R^i[|R^i| - o_1 + 1..|R^i|] = R^j[1..o_1]$ and $R^i[|R^i| - o_2 + 1..|R^i|] = R^j[1..o_2]$. Hence, $R^i[|R^i| - o_2 + 1..|R^i| - o_2 + o_1] = R^i[|R^i| - o_1 + 1..|R^i|]$. □

An even stronger result holds, namely that a shorter suffix–prefix overlap is a border (see Section 2.1) of the longer ones. A consequence of the above lemma is that the intervals of the BWT containing occurrences of R^j that overlap R^i must be nested.

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

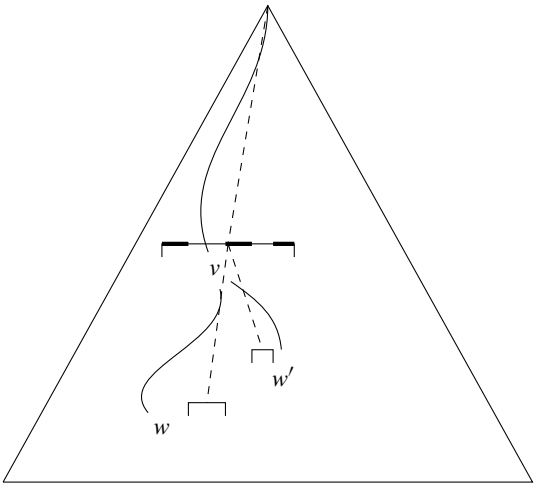


Figure 13.2 Conceptual suffix tree of read R^i associated with nested intervals in the BWT of the concatenation of reads. Curved lines denote suffix tree paths connecting the root to v, v to w, and v to w'. The corresponding associated intervals are shown and connected by dashed arrows to form the tree of nested intervals. The thick-line segments inside the interval of v show its sub-intervals to be reported.

To report only the maximal overlap, it suffices to associate each $\mathsf{SA}[r]$ with the deepest interval in the hierarchy of nested intervals that contains position r. This hierarchy can be constructed with the aid of the suffix tree of R^i as follows. For each suffix $R^i[k..|R^i|]$, we follow the path from the root to the first node v such that a prefix of the path to v spells $R^i[k..|R^i|]$ (that is, v is a proper locus of $R^i[k..|R^i|]$ using the notation introduced in Section 8.3). We store interval $[p_k^{\#}..q_k^{\#}]$ at node v. The tree with nested intervals is then constructed by linking each stored interval to its closest ancestor in the suffix tree having also an interval stored, or to the root if no such ancestor is found. This construction is illustrated in Figure 13.2.

Consider now a node v in the tree of nested intervals with a set of children $N^+(v)$. Let us denote the associated interval by $\overset{\bullet}{v}$ and the overlap length by $\mathtt{depth}(v)$. We report the overlap of length $\mathtt{depth}(v)$ between reads R^i and R^j iff $j \in [p^{\#}_{\mathtt{depth}(v)} - 1..q^{\#}_{\mathtt{depth}(v)} - 1]$ and $j \notin [p^{\#}_{\mathtt{depth}(w)} - 1..q^{\#}_{\mathtt{depth}(w)} - 1]$ for all $w \in N^+(v)$. By examining the intervals of the children, the interval $\overset{\bullet}{v}$ is easy to splice into at most $|N^+(v)| + 1$ parts containing only the entries to be reported.

THEOREM 13.3 *Let $\mathcal{R} = \{R^1, R^2, \ldots, R^d\}$ be a set of reads with overall length $\sum_{i=1}^{d} |R^i| = N$, maximum read length $\max_{i=1}^{d} |R^i| = M$, and each $R^i \in [1..\sigma]^*$. There is an algorithm to solve Problem 8.5, that is, to report all triplets (i, j, ℓ) such that a suffix of R^i overlaps a prefix of R^j by $\ell \geq \tau$ characters, where ℓ is the maximal such overlap length for the pair R^i and R^j. This algorithm uses $O(d \log N + M \log M + N \log \sigma)$ bits of space and reports all c valid triplets in $O(N \log \sigma + c)$ time.*

Proof The $O(N \log \sigma)$ time and space come from supporting backtracking on the BWT index. The array $\mathtt{RA}[1..d]$ (suffix array values only for suffixes starting with #)

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

contributes $d \log N$ bits of space. The suffix tree built for every read and the nested intervals stored in it take $O(M \log M)$ bits. The construction takes time linear in the read length; Exercise 13.11 asks you to detail the linear-time procedure to deduce the tree of nested intervals from the suffix tree. Finally, only the maximal overlaps are reported, and the value ℓ for each reported triplet (i, j, ℓ) is stored at the corresponding node in the tree of nested intervals. □

The graph constructed by maximal exact overlaps can still be considerably large, so one could think of removing arcs that can be deduced through some other vertices. We call an arc of an overlap graph *redundant* if it represents an overlap triplet $(i, j', \ell_{i,j'})$ and there also exist arcs of the overlap graph representing triplets $(i, j, \ell_{i,j})$ and $(j, j', \ell_{j,j'})$, where $\ell_{j,j'} > \ell_{i,j'}$. That is, the overlap length $\ell_{i,j'}$ between R^i and $R^{j'}$ can be deduced from overlaps of length $\ell_{i,j}$ between R^i and R^j and of length $\ell_{j,j'}$ between R^j and $R^{j'}$. After all redundant arcs have been removed, we have an *irreducible overlap graph*.

Constructing the graph and removing redundant arcs is not a good solution because the peak memory consumption is not reduced. It turns out that one can directly construct the irreducible overlap graph by extending the previous algorithm for maximal overlaps. We sketch the solution in Insight 13.3 and study the algorithm further in the exercises.

Insight 13.3 Direct construction of irreducible overlap graphs

Consider the algorithm leading to Theorem 13.3, until it reports the arcs. That is, we have at each vertex v of the tree of nested intervals the main interval $\vec{v}$ spliced into at most $|N^+(v)| + 1$ parts containing the maximal overlaps, with each such subinterval associated with the overlap length $\texttt{depth}(v)$. In addition, we replace the BWT index with the bidirectional BWT index, so that for each interval in $\mathsf{BWT}(C)$ we also know the corresponding interval in $\mathsf{BWT}(\underline{C})$; notice that the splicing on $\mathsf{BWT}(\underline{C})$ is defined uniquely by the splicing of $\mathsf{BWT}(C)$.

Let $\mathcal{I}$ collect all these (interval in $\mathsf{BWT}(C)$, interval in $\mathsf{BWT}(\underline{C})$, overlap length) triplets throughout the tree. One can then do backtracking on $\mathsf{BWT}(\underline{C})$ starting from intervals in $\mathcal{I}$ using the bidirectional BWT index, updating along all backtracking paths all intervals in $\mathcal{I}$ until they become empty on the path considered, until one finds a right extension of some interval with symbol #, or until there is only one interval left of size one. In the two latter cases, one has found a non-redundant arc, and, in the first of these two cases, the other entries in the intervals that have survived so far are redundant.

Exercise 13.12 asks you to detail the above algorithm as a pseudocode. We complement the above informal description with a conceptual simulation of the algorithm on an example. Consider the following partially overlapping reads:

```
ACGATGCGATGGCTA
      GATGGCTAGCTAG
         GGCTAGCTAGAGGTG
         GGCTAGacgagtagt
```

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

Consider the overlap search for the top-most read with the bidirectional BWT index of C = #ACGATGCGATGGCTA#GATGGCTAGCTAG#GGCTAGCTAGAGGTG#GGCTAGACGAGTAGT#. After left extensions, we know the two intervals of #GATGGCTA and #GGCTA in BWT(C). Backtracking with right extensions starts only with G, with both intervals updated into non-empty intervals corresponding to #GATGGCTAG and #GATGGCTAG; other options lead to non-empty intervals. After this, backtracking branches, updating two intervals into ones corresponding to #GATGGCTAGC and #GATGGCTAGC and into one interval corresponding to #GATGGCTAGA. The former branch continues, with the two intervals updated until the interval of #GATGGCTAGCTAG is found to have right extension with #: the non-redundant arc between the first and the second read is reported. The latter branch has one interval of size one: the non-redundant arc between the first and the last read is reported. The redundant arc between the first and the third read is correctly left unreported.

13.3 Scaffolding

In this section we are concerned with finding the relative order of, orientation of, and distance between the contigs assembled in the previous section (a *scaffolding* of the contigs). We will do this with the help of paired-end reads aligning in different contigs. In the next section we will see a method for filling the gaps between contigs.

Let us summarize below some assumptions that this scaffolding step is based on.

- Double-stranded DNA. One strand is the complement of the other, and the two strands are sequenced in opposite directions.
- Paired-end reads. Recall from Section 10.5 that a paired-end read is a pair consisting of two reads, such that the second read occurs in the genome at a known distance after the first one (in the direction of the DNA strand). We refer to the missing sequence between the occurrences of the two reads as *gap*.
- Same-strand assumption. Both reads of a paired-end read originate from the same strand, but which strand is unknown. The input paired-end reads can always be normalized to satisfy this assumption, by reverse complementing one of the two reads of a paired-end read.
- Contigs. Each assembled contig entirely originates from one unknown strand. This assumption should be guaranteed by the previous contig assembly step. However, note that this might not always be the case in practice, but such errors will be tolerated to some extent in the scaffolding formulation to be studied next.

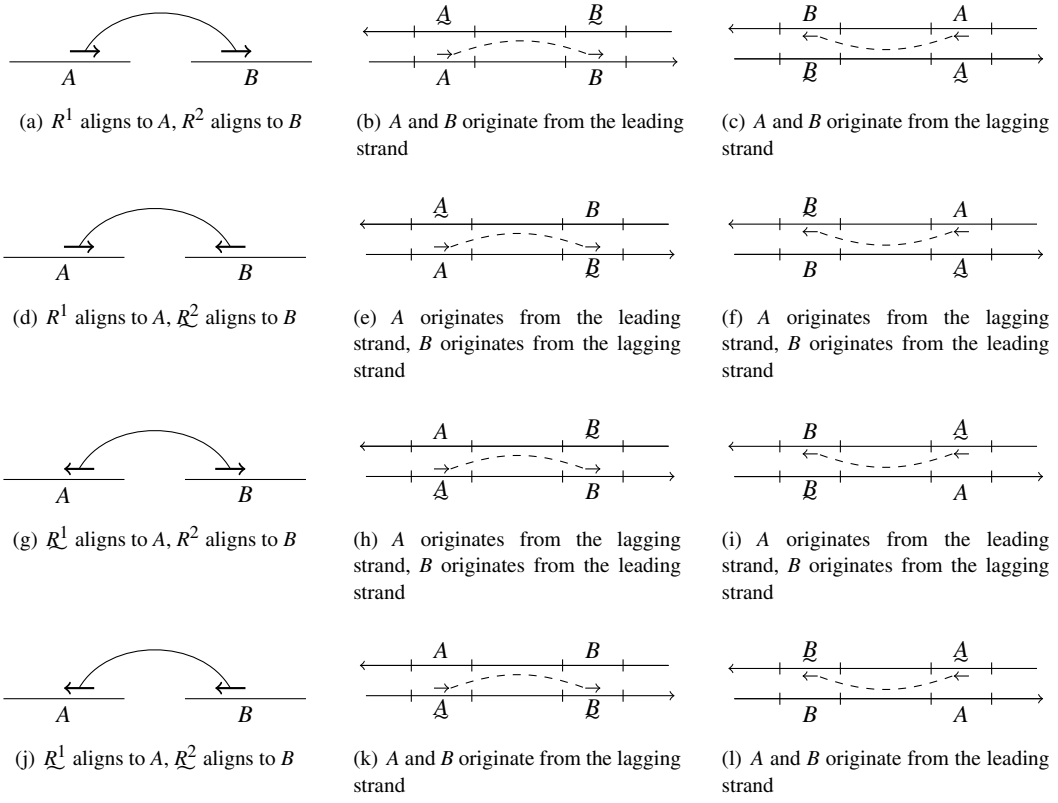
https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

To start with, assume that we have a paired-end read (R^1, R^2) with a known gap length, and assume that R^1 aligns to a position inside a contig A, and that R^2 aligns to a position inside a contig B. This alignment of (R^1, R^2) implies that

- on one strand, contig A is followed by contig B, with a gap length given by the alignment of (R^1, R^2);
- on the other strand, the reverse complement of B, denoted $\underset{\sim}{B}$, is followed by the reverse complement $\underset{\sim}{A}$ of A, with the same gap length estimate given by the alignment of (R^1, R^2).

In Figure 13.3 we show all possible alignment cases of (R^1, R^2) into contigs A and B, and then indicate the corresponding alternatives in the double-stranded DNA we are trying to reconstruct.

A scaffold will be a list of contigs appearing in the *same strand* (but which strand is unknown) of the assembled DNA. For this reason, with each contig C in a scaffold, we need to associate a Boolean variable o, which we call the *orientation* of C, such that, if $o = 0$, then we read C as given, and if $o = 1$, then we read C as reverse complemented.



(a) R^1 aligns to A, R^2 aligns to B

(b) A and B originate from the leading strand

(c) A and B originate from the lagging strand

(d) R^1 aligns to A, $\underset{\sim}{R^2}$ aligns to B

(e) A originates from the leading strand, B originates from the lagging strand

(f) A originates from the lagging strand, B originates from the leading strand

(g) $\underset{\sim}{R^1}$ aligns to A, R^2 aligns to B

(h) A originates from the lagging strand, B originates from the leading strand

(i) A originates from the leading strand, B originates from the lagging strand

(j) $\underset{\sim}{R^1}$ aligns to A, $\underset{\sim}{R^2}$ aligns to B

(k) A and B originate from the lagging strand

(l) A and B originate from the leading strand

Figure 13.3 All fours cases of aligning a paired-end read (R^1, R^2) to two contigs A and B. For each case, the two corresponding placements of the contigs in a double-stranded DNA are shown.

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

DEFINITION 13.4 *A* scaffold *S is a sequence of tuples $S = \big((C^1, o_1, d_1), (C^2, o_2, d_2), \ldots, (C^{n-1}, o_{n-1}, d_{n-1}), (C^{n-1}, o_{n-1}, d_n = 0)\big)$, where each C^i is a contig, o_i is its orientation, and d_i is the length of the gap between contigs C^i and C^{i+1} (by convention, we set $d_n = 0$).*

For simplicity, we will implicitly assume that each d_i is greater than or equal to 0, implying that in a scaffold there can be no overlap between two contigs.

In the scaffolding problem, which we will introduce in Problem 13.1, we would like to report (1) as few scaffolds as possible, while at the same time (2) making sure that a large proportion of the paired-end read alignments will be consistent with at least one scaffold. For addressing (1), we need to introduce the following notion of a *connected* scaffold with respect to a collection $\mathcal{R}$ of paired-end reads.

DEFINITION 13.5 *Given a set $\mathcal{R}$ of paired-end reads, we say that a scaffold S is* connected *with respect to $\mathcal{R}$ if the contigs of S form a connected graph G_S, in the sense that if we interpret each C^i as a vertex of a graph, and each paired-end read in $\mathcal{R}$ mapping to some contigs C^i and C^j as an edge between the vertices C^i and C^j, then we obtain a connected graph.*

For addressing (2), let us formally define this notion of consistency. For every two contigs C^i and C^j in S, we let $d(C^i, C^j)$ denote the distance between them in S, namely

$$d(C^i, C^j) = d_{\min(i,j)} + \sum_{k=\min(i,j)+1}^{\max(i,j)-1} \Big(\ell(C^k) + d_k\Big),$$

where $\ell(C^k)$ denotes the length of a contig C^k.

DEFINITION 13.6 *Given a scaffold S, a paired-end read (R^1, R^2) aligning to contigs C^i and C^j, and an integer threshold t, we say that (R^1, R^2) is* t-consistent *with S if its alignment is consistent with the cases shown in Figure 13.3, that is,*

- *if R^1 aligns to C^i and R^2 aligns to C^j, then either*
  $i < j$ and $o_i = o_j = 0$ (the case in Figure 13.3(b)), or
  $j < i$ and $o_i = o_j = 1$ (the case in Figure 13.3(c));
- *if R^1 aligns to C^i and $\underset{\sim}{R}^2$ aligns to C^j, then either*
  $i < j$ and $o_i = 0$, $o_j = 1$ (the case in Figure 13.3(e)), or
  $j < i$ and $o_i = 1$, $o_j = 0$ (the case in Figure 13.3(f));
- *if $\underset{\sim}{R}^1$ aligns to C^i and R^2 aligns to C^j, then either*
  $i < j$ and $o_i = 1$, $o_j = 0$ (the case in Figure 13.3(h)), or
  $j < i$ and $o_i = 0$, $o_j = 1$ (the case in Figure 13.3(i));
- *if $\underset{\sim}{R}^1$ aligns to C^i and $\underset{\sim}{R}^2$ aligns to C^j, then either*
  $i < j$ and $o_i = o_j = 1$ (the case in Figure 13.3(k)), or
  $j < i$ and $o_i = o_j = 0$ (the case in Figure 13.3(l)).

Moreover, we require that the distance between C^i and C^j indicated by the alignment of (R^1, R^2) differ by at most t from $d(c_i, c_j)$.

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

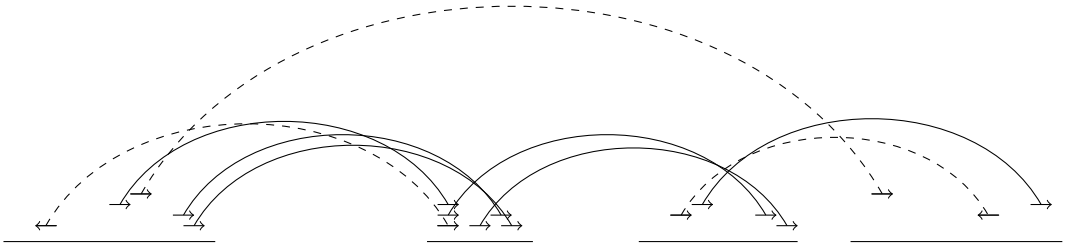


Figure 13.4 A scaffold. Solid read pairs are consistent and dashed read pairs are inconsistent. Here the distance constraint is not explicitly visualized, but the gap of each read pair is assumed to be the same. The alignments of all solid read pairs are t-consistent, with some small t, since their distance is approximately the same and their orientation is consistent with the choice made for the scaffold. The right-most dashed read pair would be consistent in terms of the distance, but the orientation differs from the choice made for the scaffold, making it inconsistent. The middle dashed read pair is inconsistent due to there being too large a distance. The left-most dashed read pair violates the same orientation constraint as the right-most gray read pair, while also the distance constraint could be violated with some choice of t.

See Figure 13.4 for an example of a connected scaffold and some consistent and inconsistent read alignments.

In practice, some paired-end reads are erroneously aligned to the contigs. Therefore, we formulate the scaffolding problem as Problem 13.1 below by asking to remove at most a given number of paired-end reads, such that the remaining alignments give rise to the least number of connected scaffolds. An alternative formulation is considered in Exercise 13.16.

Problem 13.1 Scaffolding

Given a set $\mathcal{C}$ of contigs, the alignments of a set $\mathcal{R}$ of paired-end reads in these contigs, and integers t and k, find a subset $\mathcal{R}' \subseteq \mathcal{R}$ of paired-end reads such that

- $|\mathcal{R}| - |\mathcal{R}'| \leq k$,

and, among all such subsets, $\mathcal{R}'$ has the property that the contigs in $\mathcal{C}$ can be partitioned into the minimum number of scaffolds such that

- every read in $\mathcal{R}'$ is t-consistent with some scaffold, and
- each scaffold is connected with respect to $\mathcal{R}'$.

We prove next that the scaffolding problem is NP-hard.

THEOREM 13.7 *Problem 13.1 is NP-hard.*

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

Proof We reduce from the Hamiltonian path problem. Given an undirected graph $G = (V = \{1, \ldots, n\}, E = \{e_1, \ldots, e_m\})$, we construct the following instance of the scaffolding problem. For every $i \in V$, we add a unique contig C^i to $\mathcal{C}$. For every $e_s \in E$, with $e_s = (i, j)$, we add the two paired-end reads (C^i, C^j) and (C^j, C^i) to $\mathcal{R}$. Moreover, we set the length of the gap between all $2m$ paired-end reads to some fixed non-negative integer, say 1. We prove that G has a Hamiltonian path if and only if this instance for the scaffolding problem admits a solution with $t = 0$, $k = 2m - (n - 1)$, and only one scaffold.

For the forward direction, let $P = (v_1, \ldots, v_n)$ be a Hamiltonian path in G. Then we keep in $\mathcal{R}'$ only the paired-end reads $(C^{v_i}, C^{v_{i+1}})$, for all $i \in \{1, \ldots, n-1\}$. This implies that $|\mathcal{R}| - |\mathcal{R}'| = 2m - (n-1) = k$. Moreover, the set $\mathcal{C}$ of contigs admits one scaffold, connected with respect to $\mathcal{R}'$, formed by following the Hamiltonian path P, namely $S = ((C^{v_1}, 0, 1), (C^{v_2}, 0, 1), \ldots, (C^{v_{n-1}}, 0, 1), (C^{v_n}, 0, 0))$.

For the reverse implication, let $S = ((C^{v_1}, o_{v_1}, d_{v_1}), (C^{v_2}, o_{v_2}, d_{v_2}), \ldots, (C^{v_{n-1}}, o_{v_{n-1}}, d_{n-1}), (C^{v_n}, o_{v_n}, 0))$ be the resulting scaffold. We argue that $P = (v_1, \ldots, v_n)$ is a Hamiltonian path in G, or, more precisely, that there is an edge in G between every v_i and v_{i+1}, for all $1 \leq i \leq n-1$. Because the length of the gap in any paired-end read is 1 and $t = 0$, we get that each paired-end read (C^i, C^j) is such that C^i and C^j are consecutive contigs in S. Since we added paired-end reads only between adjacent vertices in G, and the scaffold graph G_S of S is connected, P is a Hamiltonian path. □

Observe that the above proof actually shows that the problem remains hard even if it is assumed that all contigs and paired-end reads originate from the same DNA strand. Moreover, the bound k on the size $|\mathcal{R}| - |\mathcal{R}'|$ is not crucial in the proof (particularly in the reverse implication). Exercise 13.14 asks the reader to show that the scaffolding problem remains NP-hard even if no constraint k is received in the input. Finally, notice also that we allow each contig to belong to exactly one scaffold. Thus we implicitly assumed that no contig is entirely contained in a repeated region of the genome. Exercise 13.15 asks the reader to generalize the statement of Problem 13.1 to include such multiplicity assumptions.

Owing to this hardness result, we will next relax the statement of the scaffolding problem, and derive an algorithm to find at least some reasonable scaffolding, yet without any guarantee on the optimality with respect to Problem 13.1. This algorithm is similar to the minimum-cost disjoint cycle cover from Section 5.4.1. We show this mainly as an exercise on how problems from the previous chapters can elegantly deal with the complications arising from reverse complements.

We proceed in the same manner as in Insight 13.2 for reverse complements in overlap graphs. We construct an undirected graph $G = (V, E)$ by modeling each contig similarly to the approach from Insight 13.2. For every $C^i \in \mathcal{C}$, we introduce two vertices, $C^i_{\texttt{start}}$ and $C^i_{\texttt{end}}$. For every paired-end read (R^1, R^2) aligning to contigs C^i and C^j, we analogously introduce an *undirected* edge in G as follows (see Figure 13.5 and notice the analogy with the edges added in Insight 13.2):

i. if R^1 aligns to C^i and R^2 aligns to C^j, we add the edge $(C^i_{\texttt{end}}, C^j_{\texttt{start}})$;
ii. if R^1 aligns to C^i and R^2 aligns to C^j, we add the edge $(C^j_{\texttt{end}}, C^i_{\texttt{end}})$;

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

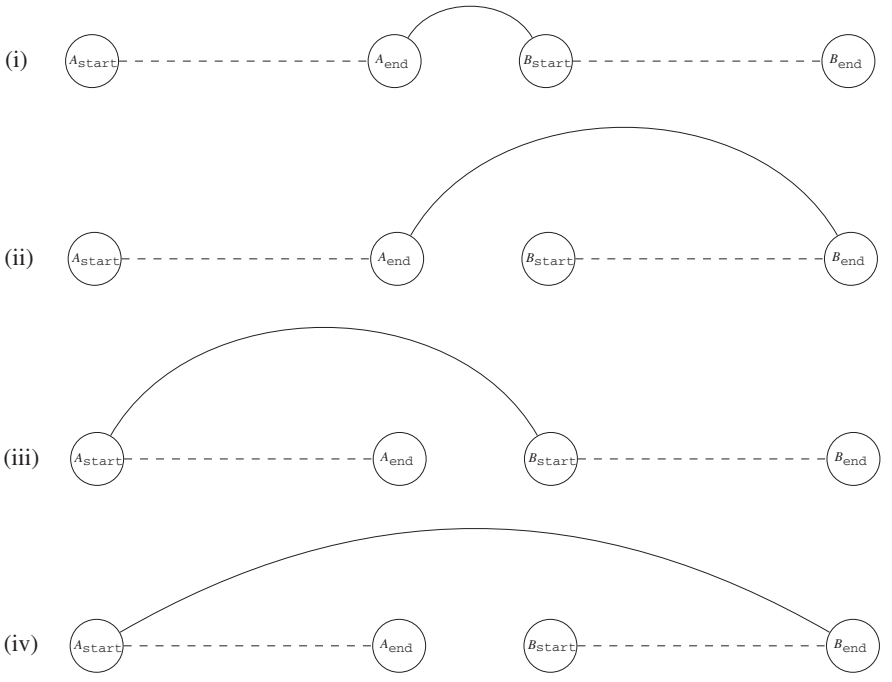


Figure 13.5 Modeling the scaffolding problem as a maximum matching problem in an undirected graph.

iii. if $\mathcal{R}^1$ aligns to C^i and R^2 aligns to C^j, we add the edge $(C^i_{\texttt{start}}, C^j_{\texttt{start}})$;
iv. if $\mathcal{R}^1$ aligns to C^i and $\mathcal{R}^2$ aligns to C^j, we add the edge $(C^i_{\texttt{start}}, C^j_{\texttt{end}})$.

We also add a cost to each edge e of G, modeling the *uncertainty* of the edge indicated by the alignments of all paired-end reads corresponding to e. This cost can be defined, for example, as the absolute deviation of the contig distances predicted by the alignments. We then find a minimum-cost maximum matching M in G (recall Exercise 5.15).

The matching M then induces a collection of scaffolds as follows. Consider the undirected graph G^* with the same vertex set as G, but whose edge set consists of M and all contig edges $(C^i_{\texttt{start}}, C^i_{\texttt{end}})$, for every $C^i \in \mathcal{C}$. Say that a path or a cycle in G^* is *alternating* if its edges alternate between contig edges and edges of M. The edges of G^* can be decomposed into a collection of alternating paths or cycles. An alternating path P in G^* naturally induces a scaffold S_P. The contigs of S_P are taken in the order in which they appear on P, oriented as given if the path first uses the `start` vertex followed by the `end` vertex, or oriented as reverse complemented if the path first uses the `end` vertex followed by the `start` vertex. The distance between consecutive contigs is taken as the average distance indicated by the alignments between the contigs. Finally, every alternating cycle can be transformed into a path by removing its edge from M with highest cost; the resulting alternating path induces a scaffold as above. See Figure 13.6 for an example.

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

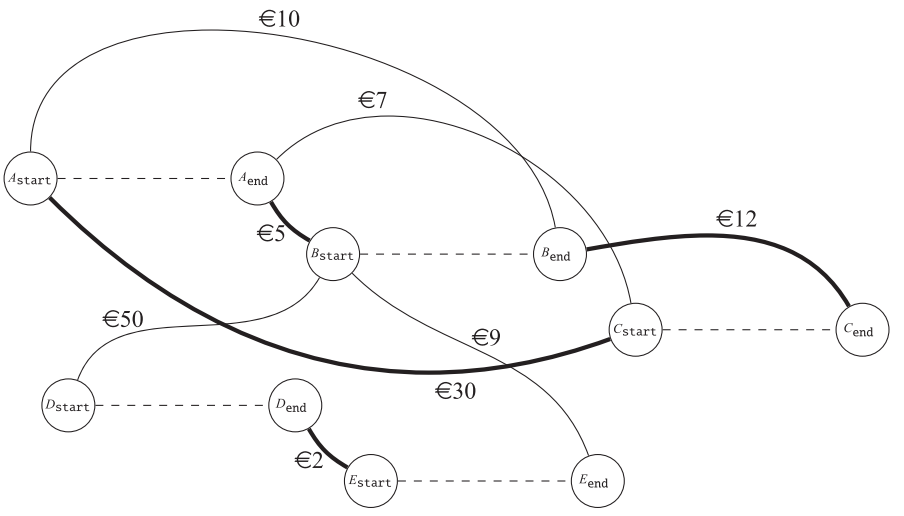


Figure 13.6 Transforming a matching (the thick edges) into scaffolds. The alternating dashed and thick edges give rise to two paths, one of which arises from a cycle cut at the highest-cost edge (€30). Two scaffolds are formed: $A \to B \to C$ and $D \to E$ (here we omit the distances between contigs).

13.4 Gap filling

In this section we are concerned with reconstructing the missing sequence between every two consecutive contigs in a scaffold. Consider the set $\mathcal{R} = \{R^1, \ldots, R^n\}$ of all reads that do not entirely align to any of the contigs. From these reads, and a pair of consecutive contigs S and T, one can build an overlap graph, and then try to find a path between the two contigs. This reconstruction phase can be guided by the constraint that the path length should match the gap length estimated in the scaffolding step. This problem is called *gap filling*.

To state this problem more formally, we let $R^0 = S$ and $R^{n+1} = T$. We build an overlap graph G having a vertex i for every R^i. For every overlap between some R^i and R^j, we add an arc (i,j). This arc is associated with the cost $c(i,j) = |R^i| - \ell_{i,j}$, where $\ell_{i,j}$ is the length of the longest suffix–prefix overlap between R^i and R^j. In other words, $c(i,j)$ is the length of the prefix of R^i obtained by removing the longest overlap with R^j. Observe that we can assume that there are no 0-cost arcs in prefix graph G, since this would indicate a read included in another read, which can be removed without changing the solution. Because of this particular definition of arc costs, this graph is also called a *prefix graph*.

A path $v_1, v_2, \ldots, v_k$ in G *spells* a string of length

$$\sum_{i=1}^{k-1} c(v_i, v_{i+1}) + |R^{v_k}|,$$

obtained by concatenating, for i from 1 to $k-1$, the prefixes of length $c(v_i, v_{i+1}) = |R^{v_i}| - \ell_{v_i,v_{i+1}}$ of R^i, followed by the last read R^{v_k}.

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

Given a path P from the start contig S to the end contig T, made up of $S = v_1, v_2, \ldots, v_k = T$, we say that the *cost* of P is

$$\mathsf{cost}(P) = \sum_{i=1}^{k-1} c(v_i, v_{i+1}),$$

namely, the cost of P is equal to the length of the string spelled by P starting with the string S, until the position immediately preceding T.

We formulate the gap-filling problem below, by requiring that the cost of a solution path belong to a given interval. In practice, d' and d should be chosen such that the midpoint $(d' + d)/2$ reflects the same distance as the length of the gap between S and T, estimated from the previous scaffolding step.

Problem 13.2 Gap filling with limited path cost

Given a prefix graph $G = (V, E)$, with a cost function $c : E \to \mathbb{Z}_+$, two of its vertices s and t, and an interval of possible path costs $[d'..d]$, decide if there is a path $P = v_1, v_2, \ldots, v_k$ such that $v_1 = s$, $v_k = t$, and

$$\mathsf{cost}(P) = \sum_{i=1}^{k-1} c(v_i, v_{i+1}) \in [d'..d],$$

and return such a path if the answer is positive.

This gap-filling problem can be solved by a dynamic programming algorithm similar to the Bellman–Ford algorithm, which we studied in Section 4.2.2, and to the Viterbi algorithm studied in Section 7.2. By using as an additional parameter the cost of a path ending in each vertex, we can decompose the problem and break possible cycles of G. More precisely, we define, for all $v \in V(G)$ and $\ell \in [0..d]$,

$$b(v, \ell) = \begin{cases} 1, & \text{if there is a path from } s \text{ to } v \text{ of cost } \textit{exactly } \ell, \\ 0, & \text{otherwise.} \end{cases}$$

We initialize $b(s, 0) = 1$, $b(v, 0) = 0$ for all $v \in V(G) \setminus \{s\}$, and $b(v, \ell) = 0$ for all $v \in V(G)$ and $\ell < 0$. The values $b(\cdot, \cdot)$ can be computed by dynamic programming using the recurrence

$$b(v, \ell) = \begin{cases} 1, & \text{if there is some in-neighbor } u \in N^-(v) \text{ such that } b(u, \ell - c(u, v)) = 1, \\ 0, & \text{otherwise,} \end{cases} \tag{13.1}$$

which we could even write more compactly as

$$b(v, \ell) = \max\{b(u, \ell - c(u, v)) \mid u \in N^-(v)\}.$$

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

The values $b(\cdot,\cdot)$ can be computed by filling a table $B[1..|V|, 0..d]$ column-by-column. This can be done in total time $O(dm)$, where m is the number of arcs of G, since, for each $\ell \in [0..d]$, each arc is inspected only once. The gap-filling problem admits a solution if there exists some $\ell \in [d'..d]$ such that $b(t, \ell) = 1$. One solution path can be traced back by repeatedly selecting the in-neighbor of the current vertex which allowed setting a value to 1 in (13.1). Therefore, we have the following result.

THEOREM 13.8 *Problem 13.2 can be solved in time $O(dm)$, where m is the number of arcs in the input prefix graph and d is the maximum path cost.*

Observe that in practice we may construct a prefix graph in which we use longest overlaps with Hamming distance at most h, where h is a threshold given in the input (this task was solved in Section 10.4). We can then associate the Hamming distance of the overlap with every arc of the input graph. In this case, we are interested in reporting, among all possible paths of cost belonging to the interval $[d'..d]$, the one with minimum total Hamming distance. Exercise 13.18 asks the reader to modify the algorithm presented here for solving this generalization, while maintaining the same time complexity.

Moreover, in this section we constructed a graph made up of *all* the reads that do not entirely align to any of the contigs. However, it makes sense to restrict the problem only to those vertices to which there is a path from s of cost at most d. Exercise 13.19 asks the reader to modify this algorithm to exploit this practical assumption.

Finally, recall that in Section 4.2.2 we interpreted the Bellman–Ford dynamic programming algorithm as a shortest-path problem on a particular layered DAG. Exercise 13.21 asks the reader to interpret the dynamic programming algorithm presented in this section as a problem in a similarly constructed DAG.

13.5 Literature

The first algorithmic approaches to genome assembly came about through the *shortest common supersequence* problem, where one tries to find the shortest sequence that has each read as its substring. This is an NP-hard problem (Räihä & Ukkonen 1981), but good approximation algorithms exist for this problem that are nicely described in a textbook (Vazirani 2001, Chapter 7). In fact, the main part of the approximation algorithms described in Vazirani (2001, Chapter 7) is very similar to the scaffolding reduction given here for turning a matching into a set of paths and cycles. Here we omitted the derivation of the approximation algorithms for the shortest common supersequence problem due to the difficulty in making such algorithms work on realistic datasets with reverse complements, measurement errors, and different alleles present.

The Eulerian path approach to sequencing by hybridization was first studied by Pevzner (1989). The NP-hardness of the problem under measurement errors is considered in Blazewicz & Kasprzak (2003). See Medvedev *et al.* (2007) and Nagarajan & Pop (2009) for applications of the Chinese postman problem to assembly.

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

Contig assembly with cleaning of substructures such as tips and bubbles is common to many assemblers based on de Bruijn graphs. One of the most popular such assemblers is described in Zerbino & Birney (2008). Read error correction is not typically described through cleaning of tips and bubbles, but quite similar methods based on k-mer indexes exist (Pevzner *et al.* 2001; Chaisson *et al.* 2004; Salmela & Schröder 2011; Yang *et al.* 2012). We used read error correction for indirectly implementing the cleaning so that static data structures could be used for contig assembly. For other applications, such as variation calling, error correction should be done more sensitively, as is partly considered in Exercise 13.2.

We used here a definition for bubbles that involves two unary parallel paths, and these are sometimes called *simple bubbles* in the literature. One can extend the definition to include a pair of non-unary paths. The algorithmics around this general definition is studied in Sacomoto (2014). A further generalization is that of *superbubbles*, introduced in Onodera *et al.* (2013). Exercise 13.8 defines a superbubble and asks the reader to write an algorithm for listing all superbubbles of a directed graph.

The overlap graph and its extension to manage with reverse complements is analogous to the ones in Kececioglu (1991) and Kececioglu & Myers (1993); the representations are not exactly the same, but they are functionally equivalent. An analogous modification for de Bruijn graphs is studied in Medvedev *et al.* (2007). The NP-hardness of many variants of the assembly problem is studied in Kececioglu (1991), Kececioglu & Myers (1993), Medvedev *et al.* (2007), and Nagarajan & Pop (2009).

For the scaffolding problem the NP-hardness of a very similar formulation was first given in Huson *et al.* (2001). As with the contig assembly and graph cleaning, many practical approaches aim to remove spurious edges in the scaffolding graph such that the scaffolding can be solved independently and even exactly on the remaining small components using solvers for integer programming formulations of the problem (Dayarian *et al.* 2010; Salmela *et al.* 2011; Lindsay *et al.* 2012). It is also possible to derive and apply fixed-parameter tractable algorithms (Gao *et al.* 2011; Donmez & Brudno 2013). Accurate estimation of the distances between consecutive contigs inside scaffolds is studied in Sahlin *et al.* (2012).

The direct and space-efficient construction of irreducible overlap graphs is from Simpson & Durbin (2010); there the resulting graph is called an *irreducible string graph*. String graphs and their reduction by removal of transitive edges were first proposed in Myers (2005).

The problem of finding paths in graphs with an exact sum of edge weights was studied in Nykänen & Ukkonen (2002). They showed that this problem is NP-hard: Exercise 13.17 asks you to rediscover this reduction. They also gave a pseudo-polynomial-time algorithm by bounding the edge weights and the requested sum. Our gap-filling problem is easier because the weights in the prefix graph are strictly positive, and therefore we could solve this restricted problem with the simple dynamic programming algorithm. This algorithm is further optimized in Salmela *et al.* (2015).

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

Exercises

13.1 Consider the following problem, called the Chinese postman problem. Given a directed graph $G = (V, E)$, a demand $d : E \to \mathbb{Z}_+$ for every arc, and a cost $c : E \to \mathbb{Q}_+$ for every arc, find a cycle in G (possibly with repeated vertices) visiting every arc (x, y) at least $d(x, y)$ times, and minimizing the sum of the costs of the arcs it traverses (note that, if the cycle uses an arc (x, y) t times, then the cost associated with this arc is $tc(x, y)$). *Hint.* Reduce the problem to a minimum-cost flow problem.

13.2 An alternative formulation of correcting reads is to directly study the k-mer spectrum. Each k-mer having unexpectedly low coverage could be marked as *erroneous*. Consider a bipartite graph with erroneous k-mers on one side and the remaining k-mers on the other side. Add all edges between the two sides and define the edge cost between two k-mers as their edit distance. Find the minimum-cost maximum matching on this graph: recall Exercise 5.14 on page 66. How can you interprete this matching in order to correct the reads? Can you find a better formulation to exploit matching or network flows for error correction?

13.3 Fix some concrete implementation capable of implementing the replacement rules related to the read error correction after detection of bubbles in an assembly graph. What time and space complexities do you achieve?

13.4 Give a linear-time algorithm to find all bubbles in a directed graph.

13.5 Modify the linear-time bubble-finding algorithm from the previous assignment to work in small space using the succinct de Bruijn graph representations of Section 9.7. *Hint.* Use a bitvector to mark the already-visited vertices. Upon ending at a vertex with no unvisited out-neighbors, move the finger onto the bitvector for the next unset entry.

13.6 Consider replacing each bubble with one of its parallel unary paths. This can introduce new bubbles. One can iteratively continue the replacements. What is the worst-case running time of such an iterative process?

13.7 Consider the above iterative scheme combined with the read error correction of Exercise 13.3. What is the worst-case running time of such an iterative read error correction scheme?

13.8 Given a directed graph $G = (V, E)$, let s and t be two vertices of G with the property that t is reachable from s, and denote by $U(s, t)$ the set of vertices reachable from s without passing through t. We say that $U(s, t)$ is a *superbubble* with *entrance* s and *exit* t if the following conditions hold:

- $U(s, t)$ equals the set of vertices from which t is reachable without passing through s;
- the subgraph of G induced by $U(s, t)$ is acyclic;
- no other vertex t' in $U(s, t)$ is such that $U(s, t')$ satisfies the above two conditions.

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

Write an $O(|V||E|)$-time algorithm for listing the entrance and exit vertices of all superbubbles in G.

13.9 Insight 13.2 introduced a special graph structure to cope with reverse complement reads in assembly graphs. Consider how maximal unary paths, tips, bubbles, and superbubbles should be defined on this special graph structure. Modify the algorithms for identifying these substructures accordingly.

13.10 Suppose you are given a set of d variable-length strings (reads) of total length N over the integer alphabet $[1..\sigma]$. Can you induce the sorted order of the strings in $O(N \log \sigma)$ time and bits of space (notice that the algorithm presented in Section 8.2.1 can sort in $O(N)$ time but uses $O(N \log N)$ bits of space)? *Hint.* Use some standard sorting algorithm, but without ever copying or moving the strings.

13.11 Recall the algorithm for computing maximal overlaps in Section 13.2.3. Fill in the details for constructing in linear time the tree of nested intervals from the suffix tree associated with intervals. *Hint*. First prove that the locus of $R^i[k..|R^i|]$ for any k is either a leaf or an internal node of depth $|R^i| - k + 1$.

13.12 Give the pseudocode for the algorithm sketched in Insight 13.3.

13.13 Analyze the running time and space of the algorithm in Insight 13.3. For this, you might find useful the concept of a *string graph*, which is like an overlap graph, but its arcs are labeled by strings as follows. An arc (i, j, ℓ) of an overlap graph becomes a bidirectional arc having two labels, $R^i[1..|R^i| - \ell]$ for the forward direction and $R^j[\ell + 1..|R^j|]$ for the backward direction. Observe that one can bound the running time of the algorithm in Insight 13.3 by the total length of the labels in the string graph before redundant arcs are removed.

13.14 Show that the following relaxation of Problem 13.1 remains NP-hard. Given a set $\mathcal{C}$ of contigs, the alignments of a set $\mathcal{R}$ of paired-end reads in these contigs, and integer t, find a subset $\mathcal{R}' \subseteq \mathcal{R}$ of paired-end reads such that $\mathcal{R}'$ has the property that the contigs in $\mathcal{C}$ can be partitioned into the minimum number of scaffolds such that

- every read in $\mathcal{R}'$ is t-consistent with some scaffold, and
- each scaffold is connected with respect to $\mathcal{R}'$.

13.15 Formulate different problem statements, similar to Problem 13.1, for taking into account the fact that a contig may belong to multiple scaffolds. What can you say about their computational complexity?

13.16 Problem 13.1 could be modified to ask for the minimum k read pairs to remove such that the remaining read pairs form a consistent scaffolding. Study the computational complexity of this problem variant.

13.17 We gave a pseudo-polynomial algorithm for gap filling (Problem 13.2), in the sense that, if the gap length parameter d is bounded (by say a polynomial function in n and m), then the algorithm runs in polynomial time (in n and m). We left open the hardness of the problem when d is unbounded. Consider a generalization of the problem

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

when the input graph is any weighted graph (with any non-negative arc costs), not just a prefix graph constructed from a collection of strings. Show that this generalized problem is NP-complete, by giving a reduction from the *subset sum* problem (recall its definition from the proof of Theorem 5.3) to this generalization (you can use some arcs with 0 cost). Why is it hard to extend this reduction to the specific case of prefix graphs?

13.18 Consider the following generalization of Problem 13.2 in which we allow approximate overlaps. Given a prefix graph $G = (V, E)$, with a cost function $c : E \to \mathbb{Z}_+$ and $h : E \to \mathbb{Z}_+$, two of its vertices s and t, and an interval of possible path costs $[d'..d]$, decide whether there is a path P made up of $s = v_1, v_2, \ldots, v_k = t$ such that

$$\mathsf{cost}(P) = \sum_{i=1}^{k-1} c(v_i, v_{i+1}) \in [d'..d],$$

and among all such paths return one P minimizing

$$\sum_{i=1}^{k-1} h(v_i, v_{i+1}).$$

Show how to solve this problem in time $O(dm)$. What is the space complexity of your algorithm?

13.19 Modify the solution to Problem 13.2 so that it runs in time $O(m_r(d + \log r))$, where r is the number of vertices reachable from s by a path of cost at most d, and m_r is the number of arcs between these vertices.

13.20 Modify the solution to Problem 13.2 so that we obtain also the number of paths from s to t whose cost belongs to the interval $[d'..d]$.

13.21 Interpret the dynamic programming algorithm given for Problem 13.2 as an algorithm working on a DAG, constructed in a similar manner to that in Section 4.2.2 for the Bellman–Ford method.

13.22 Consider a de Bruijn graph G built from a set $\mathcal{R}$ of reads sequenced from an unknown genome, and let P be a unary path in G. Under which assumptions on $\mathcal{R}$ does the unary path P spell a string occurring in the original genome?

13.23 Let $G = (V, E)$ be a directed graph. Give an $O(|V| + |E|)$-time algorithm for outputting all maximal unary paths of G.

13.24 Let $G = (V, E)$ be a directed graph, and let G' be the graph obtained from G by compacting its arcs through the following process. First, label every vertex v of G with the list $l(v) = (v)$, that is, $l(v)$ is made up only of v. Then, as long as there is an arc (u, v) such that $N^+(u) = \{v\}$ and $N^-(v) = \{u\}$ remove the vertex v, append v at the end of $l(u)$, and add arcs from u to every out-neighbor of v. Explain how, given only G', it is possible to report all maximal unary paths of G in time $O(|V| + |E|)$. What is the time complexity of computing G'?

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press

13.25 Let $G = (V, E)$ be a directed graph. A more formal way of defining a contig is to say that a path P is a *contig* if P appears as a subpath of any path in G which covers all vertices of G.

- Let P be a unary path of length at least 2. Show that P is a conting.
- Let $P = (s, v_1, \ldots, v_k, \ldots, v_n, t)$ be a path such that v_k is a unary vertex, for all $1 \leq i < k$, we have $|N^-(v_i)| = 1$, and for all $k < j \leq n$, we have $|N^+(v_i)| = 1$. Show that P is a contig.

https://doi.org/10.1017/CBO9781139940023.015 Published online by Cambridge University Press