ously for many score choices. Such methods are reviewed by Waterman [198].

The FAST implementation is described by Lipman and Pearson [127], Pearson and Lipman [155], and Pearson [153]. A thorough comparison between FASTA and pure dynamic programming was done by Pearson [154]. The BLAST program is described by Altschul et al.[12]. The idea of using common windows that can be found very quickly so that larger alignments can be built is a recurring theme. Among the various applications, we cite the work by Chao, Zhang, Ostell, and Miller [36] on local comparison between extremely long sequences. See also Chao and Miller [35] and Joseph, Meidanis, and Tiwari [104].

A number of papers discuss the relationship between distance and similarity. The standard transformation used in Section 3.6.1 was already considered by Smith, Waterman, and Fitch [176]. The review chapter by Waterman [197] devotes a section to this topic. An algorithm for alignments that allow space–space matches is given in [134].

The projections method for saving time in multiple alignment is due to Carrillo and Lipman [32]. Altschul and Lipman [13] generalized it to star alignments. Star alignments were also considered by Gusfield [83], who proved performance guarantees with respect to a sum-of-pairs distance measure. Gupta, Kececioglu, and Schäffer [81] describe their implementation of Carrillo and Lipman's algorithm, known as the Multiple Sequence Alignment (or MSA) program. Kececioglu [113] formalized the notion of "a multiple alignment that is as close as possible to a set of pairwise alignments," calling it the *maximum weight trace problem*. He proved the problem NP-hard and presented a branch-and-bound algorithm. See also [137] for an alternative, heuristic method for sequence alignment based on the notion of trace. The NP-hardness of the multiple alignment with SP measure problem and the tree alignment problem were shown by Wang and Jiang [193]. The exponential-time algorithm for tree alignment is due to Sankoff [166]. Approximation algorithms for distance-based tree alignment were described by Wang and Gusfield [192], who improved on an earlier result by Jiang, Lawler, and Wang [102]. Some heuristics for multiple alignment must solve the problem of aligning alignments. Two references on this topic are Miller [139] and Gotoh [77]. For a survey on multiple-sequence comparison methods see Chan, Wong, and Chiu [33].

Because certified methods for finding the best alignments are in practice sometimes prohibitively slow, the use of parallel computers is often regarded as a way of speeding up the process. One such attempt is described by Jones et al. [103].

Suffix trees and many other algorithms for strings are described in the book by Crochemore and Rytter [39] and in the book by Stephen [180]. Suffix arrays appeared in a paper by Manber and Myers [130]. The application of suffix trees to the primer selection problem and to the palindrome problem cited in Section 3.6.3 were given in a talk by Dan Gusfield. Gusfield is preparing a new book that will describe in depth many string algorithms and their applications in computational biology [85].

CHAPTER

4

FRAGMENT ASSEMBLY OF DNA

In Chapter 1 we saw the biological aspects of DNA sequencing. In this chapter we discuss the computational task involved in sequencing, which is called fragment assembly. The motivation for this problem comes from the fact that with current technology it is impossible to sequence directly contiguous stretches of more than a few hundred bases. On the other hand, there is technology to cut random pieces of a long DNA molecule and to produce enough copies of the pieces to sequence. Thus, a typical approach to sequencing long DNA molecules is to sample and then sequence fragments from them. However, this leaves us with the problem of assembling the pieces, which is the problem we study in this chapter. We present formal models for the problem and algorithms for its solution.

4.1 BIOLOGICAL BACKGROUND

To **sequence** a DNA molecule is to obtain the string of bases that it contains. In large-scale DNA sequencing we have a long target DNA molecule (thousands of bp) that we want to sequence. We may think of this problem as a puzzle in which we are given a double row of cards facing the table, as in Figure 4.1. We do not know which letter from the set {A, C, G, T} is written on each card, but we do know that cards in the same position of opposite strands form a complementary pair. Our goal is to obtain the letters using certain *hints*, which are (approximate) substrings of the rows. The long sequence to reconstruct is called the **target**.

In the biological problem, we know the length of the target sequence approximately, within 10% or so. It is impossible to sequence the whole molecule directly. However, we may instead get a piece of the molecule starting at a random position in one of the strands and sequence it in the canonical ($5' \rightarrow 3'$) direction for a certain length. Each such sequence is called a **fragment**. It corresponds to a substring of one of the strands of the

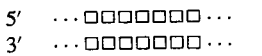

FIGURE 4.1

Unknown DNA to be sequenced.

target molecule, but we do not know which strand or its position relative to the beginning of the strand in addition it may contain errors. By using the *shotgun method* (described in Section 1.5.2), we obtain a large number of fragments and then we try to reconstruct the target molecule's sequence based on fragment overlap. Depending on experimental factors, fragment length can be as low as 200 or as high as 700. Typical problems involve target sequences 30,000 to 100,000 base-pairs long, and total number of fragments is in the range 500 to 2000.

The problem is then to deduce the whole sequence of the target DNA molecule. Because we have a collection of fragments to put together, this task is known as **fragment assembly**. We note that it suffices to determine one of the strands of the original molecule, since the other can be readily obtained because of the complementary pair rule.

In the remainder of this section we give additional details of a biological nature that are important in the design of fragment assembly algorithms.

4.1.1 THE IDEAL CASE

The best way of studying the issues involved is by looking at an example. Be aware, however, that real instances are much larger than the examples we present. Suppose the input is composed of the four sequences

```
ACCGT
CGTGC
TTAC
TACCGT
```

and we know that the answer has approximately 10 bases. One possible way to assemble this set is

```
--ACCGT--
----CGTGC
TTAC-----
-TACCGT--
TTACCGTGC
```

Notice that we aligned the input set, ignoring spaces at the extremities. We try to align in the same column bases that are equal. The only guidance to assembly, apart from the approximate size of the target, are the *overlaps* between fragments. By overlap here we mean the fact that sometimes the end part of a fragment is similar to the beginning of another, as with the first and second sequences above. By positioning fragments so that they align well with each other we get a **layout**, which can be seen as a multiple alignment of the fragments.

The sequence below the line is the **consensus sequence**, or simply *consensus,* and is the answer to our problem. The consensus is obtained by taking a majority vote among all bases in each column. In this example, every column is unanimous, so computing the consensus is straightforward. This answer has nine bases, which is close to the given target length of 10, and contains each fragment as an exact substring. However, in practice fragments are seldom exact substrings of the consensus, as we will see.

4.1.2 COMPLICATIONS

As mentioned above, real problem instances are very large. Apart from this fact, several other complications exist that make the problem much harder than the small example we saw. The main factors that add to the complexity of the problem are errors, unknown orientation, repeated regions, and lack of coverage. We describe each factor in the sequel.

Errors

The simplest errors are called *base call* errors and comprise base substitutions, insertions, and deletions in the fragments. Examples of each kind are given in Figures 4.2, 4.3, and 4.4, respectively. Transpositions are also common but we can treat them as compositions of an insertion and a deletion or two substitutions.

Input:

```
ACCGT
CGTGC
TTAC
TGCCGT
```

Answer:

```
--ACCGT--
----CGTGC
TTAC-----
-TGCCGT--
TTACCGTGC
```

FIGURE 4.2

In this instance there was a substitution error in the second position of the last fragment, where A *was replaced by* G. *The consensus is still correct because of majority voting.*

Base call errors occurs in practice at rates varying from 1 to 5 errors every 100 characters. Their distribution along the sequence is not uniform, as they tend to concentrate towards the 3′ end of the fragment. As we can see from the examples, it is still possible to reconstruct the correct consensus even in the presence of errors, but the computer program must be prepared to deal with this possibility and this usually means algorithms that require more time and space. For instance, it is possible to find the best alignments between two sequences in linear time if there are no errors, whereas we saw in Chapter 3 that quadratic algorithms are needed to account for gaps.

Apart from erroneous base calls, two other types of errors can affect assembly. One of them is the artifact of *chimeric* fragments, and the other is *contamination* by host or vector DNA. We explain each type in what follows.

Input:

```
ACCGT
CAGTGC
TTAC
TACCGT
```

Answer:

```
--ACC-GT--
----CAGTGC
TTAC------
-TACC-GT--
TTACC-GTGC
```

FIGURE 4.3

In this instance there was an insertion error in the second position of the second fragment. Base A *appeared where there should be none. The consensus is still correct because of spaces introduced in the multiple alignment and majority voting. Notice that the space in the consensus will be discarded when reporting the answer.*

Input:

```
ACCGT
CGTGC
TTAC
TACGT
```

Answer:

```
--ACCGT--
----CGTGC
TTAC-----
-TAC-GT--
TTACCGTGC
```

FIGURE 4.4

In this instance there was a deletion in the third (or fourth) base in the last fragment. The consensus is still correct because of spaces in the alignment and majority voting.

Chimeric fragments, or *chimeras*, arise when two regular fragments from distinct parts of the target molecule join end-to-end to form a fragment that is *not* a contiguous part of the target. An example is shown in Figure 4.5. These misleading fragments must be recognized as such and removed from the fragment set in a preprocessing stage.

Sometimes fragments or parts of fragments that do not have anything to do with the target molecule are present in the input set. This is due to contamination from host or vector DNA. As we saw in Section 1.5.2, the process of replicating a fragment consists of inserting it into the genome of a vector, which is an organism that will reproduce and carry along copies of our fragment. In the end, the fragment must be purified from the vector DNA, and here is where contamination occurs — if this purification is not complete. If the vector is a virus, then the infected cell — generally a bacterial cell — can also contribute some genetic material to the fragment.

Contamination is a rather common phenomenon in sequencing experiments. Witness to this is the significant quantity of vector DNA that is present in the community databases. Scientists sometimes fail to screen against the vector sequence prior to assembly and submit a contaminated consensus to the database.

As with chimeras, the remedy for this problem is to screen the data before starting assembly. The complete sequences of vectors commonly used in DNA sequencing are well known, and it is not difficult to screen all fragments against these known sequences

Input:

```
ACCGT
CGTGC
TTAC
TACCGT
TTATGC
```

Answer:

```
--ACCGT--
----CGTGC
TTAC-----
-TACCGT--
TTACCGTGC

TTA---TGC
```

FIGURE 4.5

The last fragment in this input set is a chimera. The only way to deal with chimeras is to recognize and remove them from the input set before starting assembly proper.

to see whether a substantial part of them is present in any fragment. Chimeric fragment detection leads to interesting algorithmic problems, but we will not detail them any further in this book. Pointers to relevant references are given in the bibliographic notes.

Unknown Orientation

Fragments can come from any of the DNA strands and we generally do not know to which strand a particular fragment belongs to. We do know, however, that whatever the strand the sequence read goes from 5′ to 3′. Because of the complementarity and opposite orientation of strands, the fact that a fragment is a substring of one strand is equivalent to the fact that its reverse complement is a substring of the other. As a result, we can think of the input fragments as being all approximate substrings of the consensus sought either as given or in reverse complement. Figure 4.6 shows an assembly problem involving fragments in both orientations, initially unknown, but with no errors. In practice, we have to deal with both errors and unknown orientation at the same time.

Because the orientations are unknown, in principle we should try all possible com-

Input:

```
CACGT
ACGT
ACTACG
GTACT
ACTGA
CTGA
```

Answer:

```
→ CACGT--------
→ -ACGT--------
← --CGTAGT-----
← -----AGTAC---
→ --------ACTGA
→ ---------CTGA
  CACGTAGTACTGA
```

FIGURE 4.6

Fragment assembly with unknown orientation. Initially we do not know the orientation of fragments. Each one can be used either in direct or reverse orientation. In the solution, we indicate by an arrow the chosen orientation: → means fragment as is, ← means its reverse complement.

binations, which are 2^n for a set with n fragments. Of course, this method is unacceptable and is not the way it is done in an assembly program, but it does hint at the complexity introduced by the orientation issue.

Repeated Regions

Repeated regions or **repeats** are sequences that appear two or more times in the target molecule. Figure 4.7 shows an example. Short repeats, that is, repeats that can be entirely covered by one fragment, do not pose difficulties. The worst problems are caused by longer repeats. Also, the copies of a repeat do not have to be identical to upset assembly. If the level of similarity between two copies of a repeat is high enough, the differences can be mistaken for base call errors. Remember that the assembler must be prepared to deal with errors, so there is usually some degree of tolerance in overlap detection.

FIGURE 4.7

Repeated regions. The blocks marked X_1 and X_2 are approximately the same sequence.

The kinds of problems that repeats cause are twofold. First, if a fragment is totally contained in a repeat, we may have several places to put it in the final alignment, as it may fit reasonably well in the several repeat copies. One could argue that it does not matter where we put it, since in any copy of the same repeat the consensus will be approximately the same. But the point is that, when the copies are not exactly equal, we may weaken the consensus by placing a fragment in the wrong copy.

Second, repeats can be positioned in such a way as to render assembly inherently ambiguous; that is, two or more layouts are compatible with the input fragments and approximate target length at equivalent levels of fitness. Two such cases are shown in Figures 4.8 and 4.9, respectively. The first one features three copies of the same repeat, and the second has two interleaving copies of different repeats. The common feature between these two cases is the presence of two different regions flanked by the same repeats. In the first example, both B and C are flanked by X and X. In the second, both B and D are flanked by X and Y.

So far we discussed **direct repeats**, namely, repeated copies in the same strand. However, **inverted repeats**, which are repeated regions in opposite strands, can also occur and are potentially more dangerous. As few as two copies of a long, inverted repeat are enough to make the instance ambiguous. An example is given in Figure 4.10.

Lack of Coverage

Another problem is lack of adequate coverage. We define the **coverage** at position i of the target as the number of fragments that cover this position. This concept is well

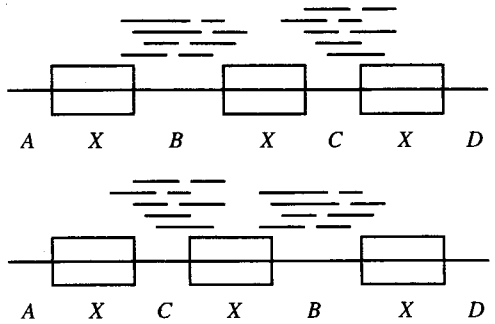

FIGURE 4.8

Target sequence leading to ambiguous assembly because of repeats of the form XXX.

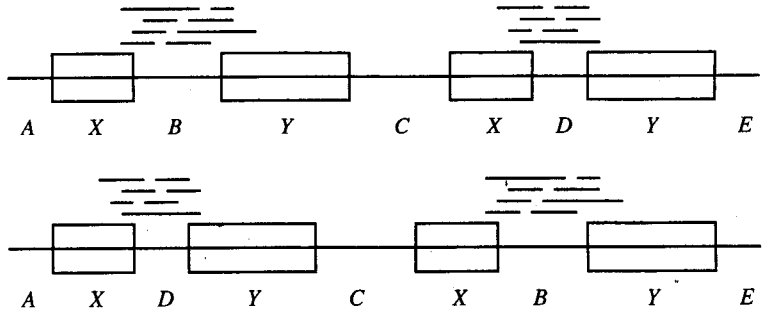

FIGURE 4.9

Target sequence leading to ambiguous assembly because of repeats of the form XYXY.

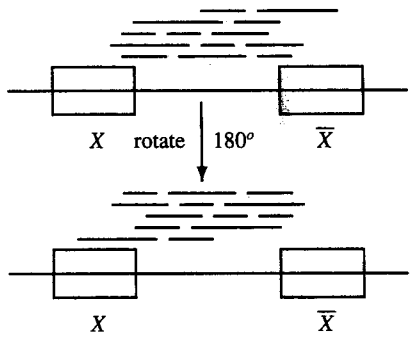

FIGURE 4.10

Target sequence with inverted repeat. The region marked $\overline{X}$ is the reverse complement of the region marked X.

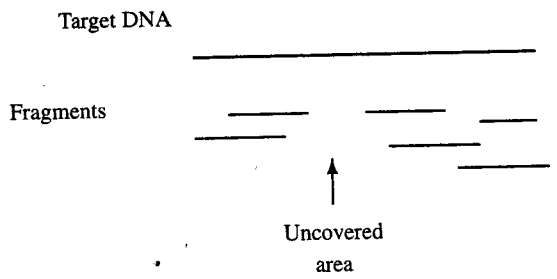

FIGURE 4.11

Example of insufficient coverage.

defined, but it is impossible to compute it because we do not know the actual positions of the fragments in the target. Even after assembly, all we have is our best guess about these positions. We can compute the *mean coverage,* though, by adding up all fragment lengths and dividing by the approximate target length.

If the coverage is equal to zero for one or more positions, then there is not enough information in the fragment set to reconstruct the target completely. Figure 4.11 shows an example in which there is an uncovered area. In such cases, the best we can hope to achieve is a layout for every one of the contiguously covered regions, called **contigs**. There are two contigs in Figure 4.11.

Lack of coverage occurs because the sampling of fragments is essentially a random process; it is therefore possible that some parts end up not being well covered or not covered at all. In general, we want not only one, but several different fragments covering any given point. The more fragments we have, the safer is our assessment of the consensus based on voting. It is also desirable to have fragments from both strands covering a given region, for it has been observed in practice that certain kinds of errors occur consistently in one strand only.

Insufficient coverage can usually be dealt with by sampling more fragments, but we must be careful with this approach. If everything is well covered except for a small portion of the target, sampling at random can be a very inefficient way of covering the gaps. An approach, called *directed sequencing* or *walking* (see Section 4.1.3), can be used in this case. To avoid the formation of these little gaps, some researchers advocate sampling at very high coverage rates, collecting fragments whose combined length is enough to cover the target molecule 8 times or more.

It is important to know how many fragments we need to generate in order to achieve a given coverage. The answer cannot be absolute, of course, since sampling is a random process, but reasonable bounds can be derived. We mention here an important result that is valid under certain simplifying assumptions.

Let T denote the length of the target molecule. Assume that all fragments have about the same length l and that we can safely recognize overlaps of at least t bases. If we sample n fragments at random, the expected number p of *apparent contigs* is given by the formula

$$p = ne^{-n(l-t)/T}. \tag{4.1}$$

The term "apparent contig" refers to the fact that we are assuming that we are able to recognize overlaps only if they have size at least t. Thus some of the contigs we have may in fact overlap by less that t bases, so they are really parts of a longer true contig but "appear" to be two separated contigs to us.

Notice that p approaches zero as the number n of fragments grows, which seems strange because we know there will always be at least one apparent contig. This happens because formula (4.1) is an approximation. It can be used also in the context of DNA mapping, where the number of contigs is much larger, so a difference of one or two is negligible. The formula is useful in the sense that it provides a reliable ballpark in a wide variety of situations.

A formula for the fraction of the target molecule covered by the fragments is also available. With the same notation as above, the fraction covered by exactly k fragments is given by

$$r_k = \frac{e^{-c}c^k}{k!}, \tag{4.2}$$

where $c = nl/T$ is the mean coverage.

4.1.3 ALTERNATIVE METHODS FOR DNA SEQUENCING

We mention here a few supplementary and alternative approaches to DNA sequencing. We start with *directed sequencing*, a method that can be used to cover small remaining gaps in a shotgun project. In direct sequencing a special primer is derived from the sequence near the end of a contig, so that fragments spanning the region including the end of this contig and the continuation of it in the target are generated. These new fragments are then sequenced and give the sequence adjacent to the contig, thereby augmenting it. Continuing in this fashion, we can cover the gap to the next contig. The problem with this approach is that it is expensive to build special primers. Also, the next step can be accomplished only after the current one, so the process is essentially sequential rather than parallel (but can be done for all gaps in parallel).

Another technique that has become very popular is called *dual end sequencing*. In a shotgun experiment several copies of the target DNA molecule are broken randomly and short pieces are selected for cloning and sequencing. We recall from Section 1.5.2 that these pieces are called *inserts* because they are inserted into a vector for amplification. Inserts sizes range from 1 to 5 kbp, but only about 200 to 700 bases can be directly read from one extremity to yield a fragment. However, it is also possible to read the other extremity if we have a suitable primer. The two fragments thus obtained belong to opposite strands, and they should be separated in the final alignment by roughly the insert size minus the fragment size. This extra information is extremely useful in closing gaps, for instance. Sometimes the dual end is sequenced only if it is necessary to close a gap. Notice that dual end sequencing exploits the fact that inserts are usually larger than the portion read from them.

A radically different approach from shotgun methods has been proposed recently. Called *sequencing by hybridization* (SBH), it consists of assembling the target molecule

based on many hybridization experiments with very short, fixed length sequences called *probes*. A hybridization simply checks whether a probe binds (by complementarity) to a DNA molecule. The idea is to design a *DNA chip* that performs all the necessary hybridizations simultaneously and delivers a list of all strings of length w, or w-mers, that exist in the target. With current technology it is possible to construct such chips for probes of length up to eight bases. Larger probe sizes seem still prohibitive at this point.

Important issues in SBH include the following. It is clear that not all target molecules can be reconstructed with a probe size as small as eight (see Exercise 10). One important problem then is to characterize the molecules that can actually be reconstructed with a given probe size. Another problem is that the hybridization experiments do not provide the number of times a given w-mer appears in the target, just whether an w-mer does appear. Clearly, it would help to have the extra information on how many copies do appear. Finally, problems with errors in the experiments and orientation need to be tackled. One proposed strategy is to reduce the shotgun data to hybridization data by simply using all the w-mers of the fragments instead of the fragments themselves and to think of them as generated by hybridization experiments. However, this strategy throws away information, because it is generally impossible to recover the fragments from the w-mers, while it is possible to generate the w-mers from the fragments. Algorithms for the SBH problem are not covered in this book, but references are given in the bibliographic notes at the end of this chapter.

4.2 MODELS

We will now examine formalisms for fragment assembly. We present in this section three models for the problem: shortest common superstring (SCS), RECONSTRUCTION, and MULTICONTIG. Each throws light on the diverse computational aspects of the problem, although none of them completely addresses the biological issues. All three assume that the fragment collection is free of contamination and chimeras.

4.2.1 SHORTEST COMMON SUPERSTRING

One of the first attempts to formalize fragment assembly was through a string problem in which we seek the shortest superstring of a collection of given strings. Accordingly, this is called the *Shortest Common Superstring* problem, or SCS. Although this model has serious shortcomings in representing the fragment assembly problem — it does not account for errors, for instance — the techniques used to tackle the resulting computational problem have application in other models as well. It is worthwhile, therefore, to study these techniques. The Shortest Common Superstring problem is defined as follows:

PROBLEM: SHORTEST COMMON SUPERSTRING (SCS)
INPUT: A collection $\mathcal{F}$ of strings.

OUTPUT: A shortest possible string S such that for every $f \in \mathcal{F}$, S is a superstring of f.

Example 4.1 Let $\mathcal{F} = \{\text{ACT}, \text{CTA}, \text{AGT}\}$. The sequence $S = \text{ACTAGT}$ is the shortest common superstring of $\mathcal{F}$. It obviously contains all fragments in $\mathcal{F}$ as substrings. To see that it is the shortest, notice that any string S' that has both $u = \text{ACT}$ and $v = \text{AGT}$ as substrings must have length at least 6. Moreover, with $|S'| = 6$ we have just the concatenations uv and vu. But CTA is a substring of $uv = S$ only.

In the context of fragment assembly, the collection $\mathcal{F}$ corresponds to the fragments, each one given by its sequence in the correct orientation, and S is the sequence of the target DNA molecule.

Notice that the computational problem specifies that S should be a perfect superstring of each fragment, not an approximate superstring, so it does not allow for experimental errors in the fragments. Furthermore, the orientation of each fragment must be known, which is seldom the case. Finally, even in a perfect assembly project in which all these factors could be somehow controlled, the shortest common superstring may not be the actual biological solution because of repeated sections in the target DNA sequence, as the following example shows.

Example 4.2 Suppose that the target molecule has two copies of an exact repeat and that fragments are sampled as shown in Figure 4.12. Notice that the repeat copies are long and contain many fragments. In this case, even if the fragments are exact substrings of the consensus, and even if we know their correct orientation, finding the shortest common superstring may not be what we want.

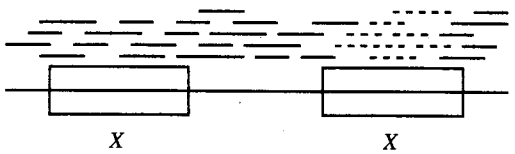

FIGURE 4.12

Target sequence with long repeat that contains many fragments. This example shows that even with no errors and known orientation, the SCS formulation fails in the presence of repeats.

Indeed, Figure 4.13 shows a different assembly, with a shorter consensus for the same fragment set. Observe that because the repeat copies are identical, a superstring may contain only one copy, which will absorb all fragments totally contained in any of the copies. The other copy can be shorter, as it must contain only fragments that cross the border between X and its flanking regions. The shortened version of X is denoted by X' in this figure.

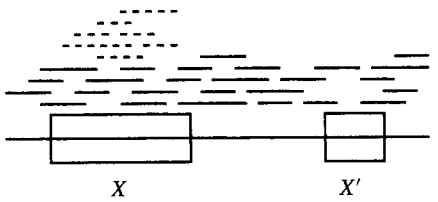

FIGURE 4.13

Alternative assembly for the fragments in the previous figure. This assembly leads to a shorter consensus, because all fragments totally contained in the rightmost copy (the dashed fragments) were moved to the leftmost copy, causing a decrease in length of the rightmost copy.

Notice that although shorter, this alternative assembly is poorer in terms of both coverage and linkage. The coverage is uneven, with many fragments spanning X and much fewer spanning X'. The linkage is poor because no fragment links the leftmost part of X' to its rightmost part. In fact, this consensus is the concatenation of two unrelated parts u and v, with u going from the beginning to about the middle of X', and v going from the middle of X' to the end. As far as shortest substrings are concerned, vu would be a perfectly equivalent consensus.

The Shortest Common Superstring problem is NP-hard, but approximation algorithms exist. However, given all the shortcomings of this model with respect to the real biological problem, such algorithms are primarily of theoretical interest.

4.2.2 RECONSTRUCTION

This model takes into account both errors and unknown orientation. To deal with errors, we need a few preliminary definitions. Recall from Chapter 3 that we can configure the basic dynamic programming sequence comparison algorithm to suit many different needs. Here we will use distance rather than similarity and a version that charges for gaps in the extremities of the second sequence only. The scoring system is edit distance; that is, one unit of distance is charged for every insertion, deletion, or substitution, except for deletion in the extremities of the second sequence, which are free of charge. The distance thus obtained will be called *substring edit distance*, to distinguish it from the classical edit distance that charges for end deletions in both strings. We denote it by d_s and define it formally by the expression

$$d_s(a, b) = \min_{s \in S(b)} d(a, s),$$

where $S(b)$ denotes the set of all substrings of b and d is the classical edit distance. Notice that substring edit distance is asymmetric, that is, in general $d_s(a, b) \neq d_s(b, a)$.


```
-----GC-GATAG----
CAGTCGCTGATCGTACG
```

FIGURE 4.14

Optimal alignment for substring edit distance, which does not charge for end deletions in the second string.

Example 4.3 If a = GCGATAG and b = CAGTCGCTGATCGTACG, then the best alignment is as indicated in Figure 4.14 and the distance is $d_s(a, b) = 2$.

Let ϵ be a real number between 0 and 1. A string f is an *approximate substring* of S at error level ϵ when

$$d_s(f, S) \leq \epsilon |f|,$$

where $|f|$ is the length of f. This means that we are allowed on average ϵ errors for each base in f. For instance, if $\epsilon = 0.05$ we are allowed five errors per hundred bases. We are now ready for the definition of fragment assembly according to the RECONSTRUCTION model.

PROBLEM: RECONSTRUCTION
INPUT: A collection $\mathcal{F}$ of strings and an error tolerance ϵ between 0 and 1.
OUTPUT: A shortest possible string S such that for every $f \in \mathcal{F}$ we have

$$\min(d_s(f, S), d_s(\overline{f}, S)) \leq \epsilon |f|,$$

where $\overline{f}$ is the reverse complement of f.

The idea is to find a string S as short as possible such that either f or its reverse complement must be an approximate substring of S at error level ϵ. This formulation will assemble correctly all the examples in Section 4.1 except for the one involving a chimeric fragment. However, the general problem is still NP-hard. This is not surprising, though, as it contains the SCS as a particular case with $\epsilon = 0$ (see Exercise 18).

In summary, RECONSTRUCTION models errors and orientation but does not model repeats, lack of coverage, and size of target.

4.2.3 MULTICONTIG

The MULTICONTIG model adds a notion of good linkage to the answer. The previous models do not care about the internal linkage of the fragments in the layout — only the final answer matters. Thus, it is also necessary to accept answers formed by several contigs, and for this reason it is called "multicontig."

We define first an error-free version of the MULTICONTIG model. Given a collection $\mathcal{F}$ of fragments, we consider a multiple alignment, or layout. This layout must be such that every column contains only one kind of base. This is where the error-free hypothesis comes in. Also, to contemplate orientation, we require that either the fragment or its reverse complement be in the alignment, but not both.

Let us call this layout $\mathcal{L}$. We number the columns from 1 to the length $|\mathcal{L}|$ of the layout. According to this numbering, each fragment f has a left endpoint $l(f)$ and a right endpoint $r(f)$, so that $|f| = r(f) - l(f) + 1$. We say that fragments f and g *overlap* in this layout if the integer intervals $[l(f)..r(f)]$ and $[l(g)..r(g)]$ intersect. The nonempty intersection $[l(f)..r(f)] \cap [l(g)..r(g)]$ is called the *overlap* between f and g. The *size* of this overlap is just the size of the intersection. Because we are using integer intervals, all sets are finite.

Among all the overlaps between fragments, we are interested in the most important ones, the ones that provide linkage. We say that an overlap $[x..y]$ is a *nonlink* if there is a fragment in $\mathcal{F}$ that properly contains the overlap on both sides, that is, if the fragment contains interval $[(x-1)..(y+1)]$. If no fragment has this property, the overlap is a *link*. The *weakest link* in a layout is the smallest size of any link in it. Finally, we say that a layout is a *t-contig* if its weakest link is at least as large as t. If it is possible to construct a t-contig from the fragments of a collection $\mathcal{F}$, we say that $\mathcal{F}$ *admits* a t-contig.

We can now formalize the fragment assembly problem according to the MULTICONTIG model. Given a collection of fragments $\mathcal{F}$ and an integer t, we want to partition $\mathcal{F}$ in the minimum number of subcollections $\mathcal{C}_i$, $1 \leq i \leq k$, such that every $\mathcal{C}_i$ admits a t-contig.

Example 4.4 Let $\mathcal{F} = \{$GTAC, TAATG, TGTAA$\}$. We want to partition $\mathcal{F}$ in the minimum number of t-contigs. If $t = 3$, this minimum is two, as follows.

```
--TAATG          GTAC
TGTAA--
```

There is no way to make one single contig with all three fragments, so two contigs is the minimum possible in this case. Notice that GTAC is its own reverse complement.

If $t = 2$, the same solution is a partition in 2-contigs, since every 3-contig is a 2-contig. However, another solution exists now:

```
TAATG---         GTAC
---TGTAA
```

Here again it is impossible to assemble $\mathcal{F}$ into one t-contig.

Finally, if $t = 1$, we see that a solution with one t-contig exists:

```
TGTAA-----
--TAATG---
------GTAC
```

A version contemplating errors can be defined as follows. The layout is not required to be error-free, but with each alignment we must associate a *consensus* sequence S of the same length, possibly containing space characters (-). The numbers $l(f)$ and $r(f)$ are still defined as the leftmost and rightmost columns of f in the alignment, but now we may have $|f| \neq r(f) - l(f) + 1$. The *image* of an aligned fragment f in the consensus is $S[l(f)..r(f)]$. Given an error tolerance degree ϵ, we say that S is an *ϵ-consensus* for this contig when the edit distance between each aligned fragment f and its image in the consensus is at most $\epsilon|f|$. We are now ready for the formal definition.

PROBLEM: MULTICONTIG
INPUT: A collection $\mathcal{F}$ of strings, an integer $t \geq 0$, and an error tolerance ϵ between 0 and 1.
OUTPUT: A partition of $\mathcal{F}$ in the minimum number of subcollections $\mathcal{C}_i$, $1 \leq i \leq k$, such that every $\mathcal{C}_i$ admits a t-contig with an ϵ-consensus.

Notice that in the RECONSTRUCTION model a fragment is required to be an approximate substring of S, but the particular place in S where it is aligned is not important. In contrast, in the MULTICONTIG formulation we need to specify explicitly where each fragment should go.

The MULTICONTIG formalization of fragment assembly is NP-hard, even in the simplest case of no errors and known orientation. It contains as a special case the problem of finding a Hamiltonian path in a restricted class of graphs.

This formulation models errors, orientation, and lack of coverage but has no provision to use information on the approximate size of the target molecule. In addition, it partially models repeats, in the sense that it can satisfactorily solve some instances with repeats, although not all of them. For instance, it can correctly reconstruct the instance given in Figure 4.12.

⋆ ALGORITHMS

4.3

In this section we present two algorithms for the case of fragments with no errors and known orientation. One of them is known as the *greedy* algorithm, and many practical systems are based on the same idea, with additions that contemplate errors and unknown orientation. The other algorithm is based on the MULTICONTIG model and is useful when we can obtain an acyclic overlap graph by discarding edges with small weight. It should be mentioned that both algorithms are of little practical value in themselves, because of the restrictive hypothesis. Nevertheless, the ideas behind them can be useful in designing algorithms that deal with real instances of the problem.

4.3.1 REPRESENTING OVERLAPS

Common superstrings correspond to *paths* in a certain graph structure based on the collection $\mathcal{F}$. We can translate properties of these superstrings to properties of the paths. Because researchers have studied paths in graphs for a long time, many people feel more comfortable dealing with graphs, and relating a new problem to them is often a good idea.

The **overlap multigraph** $\mathcal{OM}(\mathcal{F})$ of a collection $\mathcal{F}$ is the directed, weighted multigraph defined as follows. The set V of nodes of this structure is just $\mathcal{F}$ itself. A directed

edge from $a \in \mathcal{F}$ to a different fragment $b \in \mathcal{F}$ with weight $t \geq 0$ exists if the suffix of a with t characters is a prefix of b. In symbols, this is equivalent to

$$suffix(a, t) = prefix(b, t),$$

or

$$\kappa^{|a|-t} a = b\kappa^{|b|-t},$$

or

$$(a\kappa^t)b = a(\kappa^t b),$$

where κ is the killer agent defined in Section 2.1. Notice that we must have $|a| \geq t$ and $|b| \geq t$ for this edge to exist. Observe also that there may be many edges from a to b, with different values of t. This is why the structure is called a multigraph instead of simply a graph. Note that we disallow self-loops, that is, edges going from a node to itself. On the other hand, edges of zero weight are allowed.

A word must be said about edge weights. These weights represent overlaps between fragments, and we use the overlaps to join fragments into longer strings, sharing the overlapping part. This joining means that we believe that the two fragments come from the same region in the target DNA. Now this belief has to be based on solid evidence. In our case, this evidence is the size of the overlap. A short overlap provides weak evidence, whereas a long overlap gives strong evidence. When dealing with hundreds of fragments, each one a few hundred characters long, it is foolish to assume that two fragments share ends based on an overlap of, say, three characters. For this reason, sometimes we impose a minimum acceptable amount of overlap, and throw away all edges with weight below this threshold. But, in principle, we will keep all the edges in the multigraph, including the zero weight edges. There are $n(n-1)$ such edges, because any two strings share a common part of zero characters.

4.3.2 PATHS ORIGINATING SUPERSTRINGS

The overlap multigraph is important because directed paths in it give rise to a multiple alignment of the sequences that belong to this path. A consensus sequence can then be derived from this alignment, providing a common superstring of the involved sequences. Let us investigate this process more closely.

Given a directed path P in $\mathcal{OM}(\mathcal{F})$, we can construct a multiple alignment with the sequences in P as follows. Each edge $e = (f, g)$ in the path has a certain weight t, which means that the last t bases of the tail f of e are a prefix of the head g of e. Thus, f and g can be aligned, with g starting t positions before f's end.

An example is given in Figure 4.15. Only edges with strictly positive weight were drawn. The collection $\mathcal{F}$ is $\{a, b, c, d\}$, where

$$a = \text{TACGA},$$
$$b = \text{ACCC},$$
$$c = \text{CTAAAG},$$
$$d = \text{GACA}.$$

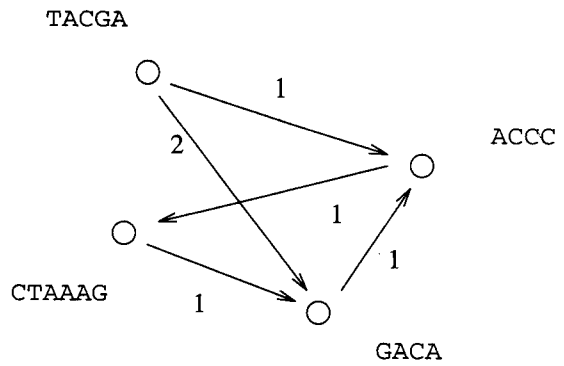

FIGURE 4.15

Overlap multigraph with zero-weight edges omitted.

Usually, we indicate paths by a list of the form vertex, edge, vertex, edge, ..., vertex, with each edge connecting its two neighboring vertices in the list. However, in this example there are no parallel edges, so, for simplicity, we will indicate the paths by just a list of the vertices involved. The path $P_1 = dbc$ leads to the following alignment:

```
GACA--------
---ACCC-----
------CTAAAG
```

whereas path $P_2 = abcd$ originates the following alignment.

```
TACGA-----------
----ACCC--------
-------CTAAAG---
------------GACA
```

In general, let P be a path in $\mathcal{OM}(\mathcal{F})$, and let A be the set of fragments involved in P. Paths by definition cannot repeat vertices, so we have exactly $|A| - 1$ edges in P. The common superstring derived from P as above will be called $S(P)$.

The relationship between the total length of A, the path's weight, and the superstring length is given by the following equation.

$$\|A\| = w(P) + |S(P)|, \tag{4.3}$$

where $\|A\| = \sum_{a \in A} |a|$ is the sum of lengths of all sequences in A, and $w(P)$ is the weight of P. Is this really true? Let us check it for some simple cases.

Start with a path with just one fragment and no edges. Then $A = \{f\}$, $\|A\| = |f|$, and $w(P) = 0$, since P has no edges. Furthermore, $S(P)$ is simply f. It is easy to see that Equation (4.3) holds in this case. Let us try something larger. Suppose that $P = f_1 e_1 f_2 e_2 \cdots f_l$ and that the formula holds for P. What happens if we consider an additional edge? We have $P' = P e_{l+1} f_{l+1}$ and

$$w(P') = w(P) + w(e_{l+1}) = w(P) + k,$$

if $w(e_{l+1}) = k$.

On the other hand, $S(P') = S(P)suffix(f_{l+1}, |f_{l+1}| - k)$, because the first k characters of f_{l+1} are already present in $S(P)$. We need to concatenate the remaining suffix of f_{l+1} only. Then,

$$\begin{aligned} |S(P')| &= |S(P)| + |f_{l+1}| - k \\ &= \|A\| - w(P) + |f_{l+1}| - k \\ &= \|A'\| - w(P') \end{aligned}$$

and the formula holds. We have just proved Equation (4.3) by induction.

Another way of seeing it is by using the killer character. If $P = f_1 e_1 f_2 e_2 \ldots f_l$,

$$S(P) = f_1 \kappa^{w(e_1)} f_2 \kappa^{w(e_2)} \cdots \kappa^{w(e_{l-1})} f_l.$$

The right-hand side is not ambiguous in this case because

$$(f_i \kappa^{w(e_i)}) f_{i+1} = f_i (\kappa^{w(e_i)} f_{i+1}),$$

by definition of edge weight.

Computing the length,

$$\begin{aligned} |S(P)| &= |f_1| - w(e_1) + |f_2| - w(e_2) + \cdots - w(e_{l-1}) + |f_l| \\ &= \sum_{i=1}^{l} |f_i| - \sum_{i=1}^{l-1} w(e_i) \\ &= \|A\| - w(P). \end{aligned}$$

So, every path originates a common superstring of the fragments involved. This is particularly important when we have a path that goes through every vertex. Such paths are called *Hamiltonian paths* (see Section 2.2). If we get a Hamiltonian path, we have a common superstring of $A = \mathcal{F}$. In this case Equation 4.3 becomes

$$|S(P)| = \|\mathcal{F}\| - w(P) \tag{4.4}$$

and, since $\|\mathcal{F}\|$ is constant, that is, independent of the particular Hamiltonian path we take, we see that minimizing $|S(P)|$ is equivalent to maximizing $w(P)$.

4.3.3 SHORTEST SUPERSTRINGS AS PATHS

We have seen that every path corresponds to a superstring. Is the converse true? Not always, as the following example shows. For the fragment set that originated the graph in Figure 4.15, the superstring

GTATACGACCCAAACTAAAGACAGGG

does not correspond to any path, and it easy to see why. Superstrings may contain unnecessary characters not present in any fragment.

But shortest superstrings cannot waste any characters. So, do shortest superstrings always correspond to paths? The answer is affirmative, and we will prove this result in the sequel. We first observe that it is true for certain collections that are *substring-free,* defined below, and then generalize the result to any collection.

A collection $\mathcal{F}$ is said to be substring-free if there are no two distinct strings a and b in $\mathcal{F}$ such that a is a substring of b. The advantage of dealing with substring-free collections is that a partial converse of the path-to-superstring construction is easier to prove, as illustrated by the following result.

THEOREM 4.1 Let $\mathcal{F}$ be a substring-free collection. Then for every common superstring S of $\mathcal{F}$ there is a Hamiltonian path P in $\mathcal{OM}(\mathcal{F})$ such that $S(P)$ is a subsequence of S.

Proof. The string S is a common superstring of $\mathcal{F}$, so for each $f \in \mathcal{F}$ there is an interval $[l(f)..r(f)]$ of S such that $S[l(f)..r(f)] = f$. For some fragments there may be many positions in which to anchor them. Never mind. Take one of them and fix it for the rest of this proof.

No interval of the form $[l(f)..r(f)]$ is contained in another such interval, because $\mathcal{F}$ is substring-free. These intervals have the following important property: All left endpoints are distinct and all right endpoints are distinct, although there may be left endpoints equal to right endpoints. Moreover, if we sort the fragments in increasing order of left endpoint, the right endpoints end up in increasing order as well. Indeed, if we had two strings f and g in $\mathcal{F}$ with $l(f) \leq l(g)$ and $r(f) \geq r(g)$, then the interval of g would be contained in the interval of f, which is impossible. So, we have the intervals sorted by increasing initial *and* final endpoints. Let $(f_i)_{i=1,\ldots,m}$ be this ordering. The fragments in this order make up the claimed path.

Consider the relationship between f_i and f_{i+1}, for each i such that $1 \leq i \leq m-1$. If $l(f_{i+1}) \leq r(f_i)+1$, then there is an edge in $\mathcal{OM}(\mathcal{F})$ with weight $r(f_i)+1-l(f_{i+1})$. If $l(f_{i+1}) > r(f_i)+1$, then take the edge of weight zero from f_i to f_{i+1}. The characters $S[j]$ for $r(f_i) + 1 \leq j \leq l(f_{i+1}) - 1$ will be discarded, because they do not belong to any interval and are therefore superfluous as far as making a common superstring is concerned. These characters are in S but not in $S(P)$, which is therefore a subsequence of S. ■

COROLLARY 4.1 Let $\mathcal{F}$ be a substring-free collection. If S is a shortest common superstring of $\mathcal{F}$, there is a Hamiltonian path P such that $S = S(P)$.

Proof. By the previous theorem, there is such a path with $S(P)$ a subsequence of S. But $S(P)$ is also a common superstring, because P is Hamiltonian. Because S is a *shortest* common superstring, we have $|S(P)| \geq |S|$ and therefore $S = S(P)$. ■

We now generalize this result to any collection. First, a few definitions are in order. A collection of strings $\mathcal{F}$ *dominates* another collection $\mathcal{G}$ when every $g \in \mathcal{G}$ is a substring of some $f \in \mathcal{F}$. For instance, if $\mathcal{G} \subseteq \mathcal{F}$, then $\mathcal{F}$ dominates $\mathcal{G}$. Two collections $\mathcal{F}$ and $\mathcal{G}$ are said to be *equivalent*, denoted $\mathcal{F} \equiv \mathcal{G}$, when $\mathcal{F}$ dominates $\mathcal{G}$ and $\mathcal{G}$ also dominates $\mathcal{F}$. Equivalent collections have the same superstrings, so this notion is important for us.

LEMMA 4.1 Two equivalent substring-free collections are identical.

Proof. If $\mathcal{F} \equiv \mathcal{G}$ and $\mathcal{F}$ is substring-free, then $\mathcal{F} \subseteq \mathcal{G}$. To see that, consider $f \in \mathcal{F}$. Since $\mathcal{F}$ is dominated by $\mathcal{G}$, there must be $g \in \mathcal{G}$ such that f is a substring of g. But $\mathcal{F}$

also dominates $\mathcal{G}$, so there is a superstring h of g in $\mathcal{F}$. By transitivity, f is a substring of h. But $\mathcal{F}$ is substring-free, so $f = h$, and, since g is "in between" them, $f = g = h$. It follows that $f = g \in \mathcal{G}$. Since f is arbitrary, we conclude that $\mathcal{F} \subseteq \mathcal{G}$. The same argument with $\mathcal{F}$ and $\mathcal{G}$ interchanged shows that $\mathcal{G} \subseteq \mathcal{F}$. Hence, $\mathcal{F} = \mathcal{G}$. ■

The next theorem helps us understand how to obtain a substring-free collection from an arbitrary collection of fragments.

THEOREM 4.2 Let $\mathcal{F}$ be a collection of strings. Then there is a unique substring-free collection $\mathcal{G}$ equivalent to $\mathcal{F}$.

Proof. Uniqueness is guaranteed by the previous lemma. Now let us turn to the existence. We want to show that for every $\mathcal{F}$ there is a substring-free collection $\mathcal{G}$ equivalent to $\mathcal{F}$. We do that by induction on the number of strings in $\mathcal{F}$.

For $|\mathcal{F}| = 0$, $\mathcal{F}$ itself is substring-free, so we are done. For $|\mathcal{F}| \geq 1$, $\mathcal{F}$ may or may not be substring-free. If it is, we are again done. If it is not, then there are a and b in $\mathcal{F}$ such that a is a substring of b. Remove a from $\mathcal{F}$, obtaining a smaller collection $\mathcal{F}' = \mathcal{F} - \{a\}$. By the induction hypothesis, $\mathcal{F}'$ has an equivalent substring-free collection $\mathcal{G}$. But it is easy to verify that $\mathcal{F} \equiv \mathcal{F}'$. Obviously $\mathcal{F}$ dominates $\mathcal{F}'$, which is a subset of $\mathcal{F}$. On the other hand, the only string that is in $\mathcal{F}$ but not in $\mathcal{F}'$ is a, which is covered by $b \in \mathcal{F}'$. Hence, $\mathcal{G}$ is also equivalent to $\mathcal{F}$ and we are done. ■

This result tells us that if we are looking for common superstrings, we might as well restrict ourselves to substring-free collections, because given any collection there is a substring-free one equivalent to it, that is, having the same common superstrings. Incidentally, the proof of Theorem 4.2 also gives us a way of obtaining the unique substring-free collection equivalent to a given $\mathcal{F}$: Just remove from $\mathcal{F}$ all strings that are substrings of other elements of $\mathcal{F}$.

4.3.4 THE GREEDY ALGORITHM

We now know that looking for shortest common superstrings is the same as looking for Hamiltonian paths of maximum weight in a directed multigraph. Moreover, because our goal is to maximize the weight, we can simplify the multigraph and consider only the heaviest edge between every pair of nodes, discarding other, parallel edges of smaller weight. Any path that does not use the heaviest edge between a pair of nodes can be improved, so it is not a maximum weight path. Let us call this new graph the *overlap graph* of $\mathcal{F}$, denoted by $\mathcal{OG}(\mathcal{F})$. In an actual implementation we would not construct these edges one by one. Using suffix trees (see Section 3.6.3), we can build the edges simultaneously, saving both space and time in the computer (see the bibliographic notes). The same structures can help in deciding which fragments are substrings of other fragments and may therefore be left out of the graph.

The following algorithm is a "greedy" attempt at computing the heaviest path. The basic idea employed in it is to continuously add the heaviest available edge, which is one that does not upset the construction of a Hamiltonian path given the previously chosen edges. Because the graph is complete, that is, there are edges between every pair of nodes, this process stops only when a path containing all vertices is formed.

In a Hamiltonian path, or in any path for that matter, we cannot have two edges leaving from the same node, or two edges leading to the same node. In addition, we have to prevent the formation of cycles. So these are the three conditions we have to test before accepting an edge in our Hamiltonian path. Edges are processed in nonincreasing order by weight, and the procedure ends when we have exactly $n - 1$ edges, or, equivalently, when the accepted edges induce a connected subgraph.

Figure 4.16 shows the algorithm. For each node, we keep its current indegree and outdegree with respect to the accepted edges. This information is used to check whether a previously chosen edge has the same tail or head as the currently examined edge. In addition, we keep the disjoint sets that form the connected components of the graph induced by the accepted edges. It is easy to see that a new edge will form a cycle if and only if its tail and head are in the same component. Therefore, this structure is needed to check for cycles. The disjoint sets are maintained by a disjoint-set forest data structure (mentioned in Section 2.3).

We described the greedy algorithm in terms of graphs, but it can be implemented using the fragments directly. The algorithm corresponds to the following procedure being continuously applied to the collection of fragments, until only one fragment remains. Take the pair (f, g) of fragments with largest overlap k, remove the two fragments from $\mathcal{F}$, and add $f\kappa^k g$ to $\mathcal{F}$. We assume that the collection $\mathcal{F}$ is substring-free.

This greedy strategy does not always produce the best result, as shown by the following example.

```
Algorithm Greedy
    input: weighted directed graph OG(F) with n vertices
    output: Hamiltonian path in OG(F)
    // Initialize
    for i ← 1 to n do
        in[i] ← 0 // how many selected edges enter i
        out[i] ← 0 // how many selected edges exit i
        MakeSet(i)
    // Process
    Sort edges by weight, heaviest first
    for each edge (f, g) in this order do
        // test edge for acceptance
        if in[g] = 0 and out[f] = 0 and FindSet(f) ≠ FindSet(g)
            select (f, g)
            in[g] ← 1
            out[f] ← 1
            Union(FindSet(f), FindSet(g))
        if there is only one component
            break
    return selected edges
```

FIGURE 4.16

Greedy algorithm to find Hamiltonian path.

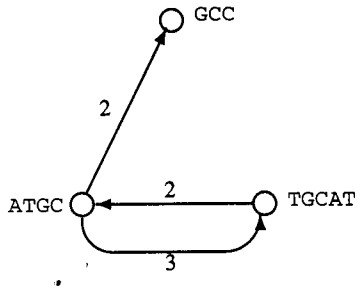

FIGURE 4.17

A graph where the greedy algorithm fails.

Example 4.5 Suppose we have

$$\mathcal{F} = \{\texttt{GCC}, \texttt{ATGC}, \texttt{TGCAT}\}.$$

The overlap graph looks like Figure 4.17, where we omitted the zero edges to avoid cluttering the drawing.

In this graph, the edge of weight 3 is the first one to be examined and is obviously accepted because the first edge is always accepted. However, it invalidates the two edges of weight 2, so the algorithm is forced to select an edge of weight zero to complete the Hamiltonian path. The two rejected edges form a path of total weight 4, which is the heaviest path in this example.

This example shows that the greedy algorithm will not always return the shortest superstring. Is there an algorithm that works in all cases? Apparently not. We are trying to solve the SCS problem through the Hamiltonian path (HP) problem; but as we have seen in Section 2.3, the HP problem is NP-complete. Perhaps we could have better luck trying another approach to solve the SCS problem. Unfortunately this is probably not the case either, since, as already mentioned, it can be shown that the SCS problem is NP-hard.

4.3.5 ACYCLIC SUBGRAPHS

The hardness results we mentioned in the previous section apply to an arbitrary collection of fragments. In this section we consider the problem of assembling fragments without errors and known orientation assuming that the fragments have been obtained from a "good sampling" of the target DNA. As shown in the sequel, this assumption enables us to derive algorithms that deliver the correct solution and are particularly efficient.

What do we mean by a "good sampling"? Basically, we want the fragments to cover the entire target molecule, and the collection as a whole to exhibit enough linkage to guarantee a safe assembly. Recall from Section 4.2.3 the definition of t-contig. Our definitions here are based on similar concepts.

Suppose that S is a string over the alphabet $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. Recall from Section 2.1 that an *interval* of S is an integer interval $[i..j]$ such that $1 \leq i \leq j+1 \leq |S|+1$. A *sampling* of S is a collection $\mathcal{A}$ of intervals of S. The sampling $\mathcal{A}$ *covers* S if for any i such that $1 \leq i \leq |S|$ we have at least one interval $[j..k] \in \mathcal{A}$ with $i \in [j..k]$.

We say that two intervals α and β are *linked at level* t if $|\alpha \cap \beta| \geq t$. The entire sampling $\mathcal{A}$ is said to be *connected at level* t if for every two intervals α and β in $\mathcal{A}$ there is a series of intervals α_i for $0 \leq i \leq l$ such that $\alpha = \alpha_0$, $\beta = \alpha_l$ and α_i is linked to α_{i+1} at level t for $0 \leq i \leq l-1$. Note that we are using parameter t to measure how *strong* a link is. A sampling $\mathcal{A}$ will be considered good if it is connected at level t.

Our goal is to study fragment collections coming from connected at level t samplings that cover a certain string S. The value of t is around 10 in typical, real instances, but in our study we will not fix it. We do assume that t is a nonnegative integer, though.

The fragment collection *generated* by a sampling $\mathcal{A}$ is

$$S[\mathcal{A}] = \{S[\alpha] \mid \alpha \in \mathcal{A}\}.$$

We say that a sampling $\mathcal{A}$ is *subinterval-free* if there are no two intervals $[i..j]$ and $[k..l]$ in $\mathcal{A}$ with $[i..j] \subseteq [k..l]$. The following observation about subinterval-free samplings will be coming into play again and again. In a subinterval free sampling, no two intervals can have the same left endpoint. Otherwise, whichever has the larger right endpoint would contain the other. An analogous argument holds for right endpoints.

We consider also modified forms of our overlap graphs. Given a collection $\mathcal{F}$ and a nonnegative integer t define $\mathcal{OM}(\mathcal{F}, t)$ as the graph obtained from $\mathcal{OM}(\mathcal{F})$ by keeping only the edges of weight at least t. A similar construction defines $\mathcal{OG}(\mathcal{F}, t)$. This reduces the overall number of edges, keeping only the stronger ones.

We will use the symbol $\alpha \xrightarrow{w} \beta$ to mean that $|\alpha| \geq w$, $|\beta| \geq w$, and $l(\beta) + w - 1 = r(\alpha)$. Similarly, $f \xrightarrow{w} g$ means that there is an edge from f to g in $\mathcal{OM}(\mathcal{F})$ of weight w.

Suppose now that we got such a collection $\mathcal{F}$ and would like to recover S. How can we do that? What are the good algorithms for it? It turns out that the answer depends strongly on the repeat structure of S. If S does not have any repeats of size t or more, then it is easy to reconstruct it. But if S has such repeats, the problem gets more complex, and in some cases S cannot be recovered without ambiguity.

One thing that is independent of the repeat structure of S is the existence of a Hamiltonian path in the graph $\mathcal{OM}(\mathcal{F}, t)$. We state and prove this basic result here, before embarking on an analysis of the repeat structure of S. First, an auxiliary lemma.

LEMMA 4.2 Let S be a string over $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ and $\mathcal{A}$ be a subinterval-free, connected at level t sampling of S, for some $t \geq 0$. If α is an interval in $\mathcal{A}$ such that there is another interval β in $\mathcal{A}$ with $l(\alpha) < l(\beta)$, then β can be chosen so that

$$r(\alpha) + 1 - l(\beta) \geq t.$$

Proof. We know $\mathcal{A}$ is connected at level t, therefore there is a sequence $(\alpha_i)_{0 \leq i \leq l}$ of intervals in $\mathcal{A}$ such that $\alpha = \alpha_0$, $\beta = \alpha_l$, and $|\alpha_i \cap \alpha_{i+1}| \geq t$ for $0 \leq i \leq l-1$.

Let i be the smallest index such that $l(\alpha) < l(\alpha_i)$. Because $\beta = \alpha_l$ satisfies this property, we know that $i \leq l$. On the other hand, $i > 0$ because $\alpha = \alpha_0$ does not satisfy the property. Because i is the smallest index with the property, we have $l(\alpha_{i-1}) < l(\alpha) < l(\alpha_i)$. It follows that $r(\alpha_{i-1}) < r(\alpha) < r(\alpha_i)$, since $\mathcal{A}$ is subinterval-free. The picture looks like Figure 4.18.