Algorithmica (1995) 13: 7–51

Algorithmica
© 1995 Springer-Verlag New York Inc.

Combinatorial Algorithms for DNA Sequence Assembly[1]

J. D. Kececioglu[2] and E. W. Myers[3]

Abstract. The trend toward very large DNA sequencing projects, such as those being undertaken as part of the Human Genome Program, necessitates the development of efficient and precise algorithms for assembling a long DNA sequence from the fragments obtained by shotgun sequencing or other methods. The sequence reconstruction problem that we take as our formulation of DNA sequence assembly is a variation of the shortest common superstring problem, complicated by the presence of sequencing errors and reverse complements of fragments. Since the simpler superstring problem is NP-hard, any efficient reconstruction procedure must resort to heuristics. In this paper, however, a four-phase approach based on rigorous design criteria is presented, and has been found to be very accurate in practice. Our method is robust in the sense that it can accommodate high sequencing error rates, and list a series of alternate solutions in the event that several appear equally good. Moreover, it uses a limited form of multiple sequence alignment to detect, and often correct, errors in the data. Our combined algorithm has successfully reconstructed nonrepetitive sequences of length 50,000 sampled at error rates of as high as 10%.

Key Words. Computational biology, Branch-and-bound algorithms, Approximation algorithms, Fragment assembly, Sequence reconstruction.

1. Introduction.

DNA sequences may be viewed abstractly as strings over the four-letter alphabet {a, c, g, t}, each letter standing for the first character of the chemical name of the *nucleotide* comprising the polymer's chain. Current gel electrophoresis technology permits experimentalists to determine directly the sequence of a DNA strand 300–700 nucleotides in length. Determining the sequence of a longer strand, of say 10,000–100,000 nucleotides, requires an indirect approach. In the *shotgun sequencing* method, the experimentalist randomly samples fragments of a length short enough to be determined by electrophoresis. Whenever two fragments are sampled from regions that intersect, this is detected as an overlap in the sequences of the fragments. With sufficient sampling the underlying sequence can eventually be reconstructed by *assembling* the fragments according to their overlaps. Our problem is to perform the assembly of the current fragment set at any point in such a project.

[1] This research was supported by the National Library of Medicine under Grant R01-LM4960, by a postdoctoral fellowship from the Program in Mathematics and Molecular Biology of the University of California at Berkeley under National Science Foundation Grant DMS-8720208, and by a fellowship from the Centre de recherches mathématiques of the Université de Montréal.

[2] Department of Computer Science, The University of Georgia, Athens, GA30602, USA. kece@cs.uga.edu.

[3] Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA. gene@cs.arizona.edu.

Received October 19, 1992; revised February 8, 1993. Communicated by E. W. Myers.

This seemingly simple procedure is made difficult by several exacerbating factors. First, the fragments may not assemble into a single reconstruction due to insufficient coverage of the underlying sequence. Second, errors are present in the fragment sequences due to experimental errors in the electrophoresis procedure. With current technology, anywhere from 0.5% to 5% of the sequence of a fragment may be incorrect. Third, an overlap may not be due to the fact that the fragment intervals intersect, but may simply be due to chance. In a project involving a thousand fragments, given the presence of error, such spurious overlaps occur. Finally, DNA is double-stranded, and a particular fragment may have come from either strand. Hence a fragment may represent the sequence on one strand, or the reverse complement sequence on the opposite strand. In this case we say the *orientation* of the fragments is not known.

This paper develoops an algorithm for sequence assembly in the most general setting, with incomplete coverage, sequencing errors, unknown fragment location, and unknown fragment orientation. As the error in the fragments decreases, the speed of the algorithm increases. It can also accommodate information concerning fragment order and orientation, and generate alternate solutions on demand. For the subproblems that arise, we either design exact algorithms that find an optimal solution but may take exponential time, or approximation algorithms that find a solution close to optimal and always run fast. For some problems, we design both.

To define the problem formally, let us denote the minimum number of insertions, deletions, and substitutions that are required to edit sequence A into sequence B by $d(A, B)$, the *edit distance* between A and B. We denote the *reverse complement* of sequence A by $\overleftarrow{A}$. Sequence $\overleftarrow{A}$ is obtained by reversing A and mapping each character to its complement. Writing $\bar{a}$ for the complement of character a, in the DNA alphabet $\bar{\mathsf{a}} = \mathsf{t}$, $\bar{\mathsf{t}} = \mathsf{a}$, $\bar{\mathsf{c}} = \mathsf{g}$, and $\bar{\mathsf{g}} = \mathsf{c}$. If $A = \mathsf{gacct}$, for example, $\overleftarrow{A} = \overline{\mathsf{tccag}} = \mathsf{aggtc}$.

Under the principle of parsimony, a natural formulation of the sequence assembly problem is to determine a shortest sequence that explains all of the fragments. Formally, this is the following.

Definition. The *DNA sequence reconstruction problem* (Reconstruct) is, given a collection $\mathscr{F}$ of fragment sequences and an error rate $0 \le \varepsilon < 1$, find a shortest sequence S such that for every fragment $A \in \mathscr{F}$ there is a substring B of S such that

$$\min(d(A, B), d(\overleftarrow{A}, B)) \le \varepsilon |A|.$$

In the related *shortest common superstring problem* (Superstring) we are given a collection of strings and seek a shortest string S, called a superstring, such that every member of the collection is a substring of S. In essence Reconstruct is a shortest common superstring problem where a fragment is considered to match the superstring if the fragment, or its reverse complement, can be aligned to the superstring within a length-relative error threshold of ε. In fact, Superstring can be reduced to the sequence reconstruction problem with $\varepsilon = 0$. Since Superstring is NP-complete [13], this implies that Reconstruct is NP-complete. Details may be found in [18].

Related Work. Prior work related to DNA sequence assembly may be grouped by three categories. In the first group of papers. Shapiro [32], Hutchinson [17], Smetanič and Polozov [33], Gallant [12], and Foulser [7] examine an early model of the problem where fragments do not contain errors and are partitioned into classes such that concatenating the fragments in a class, in some order, gives the underlying sequence. The problem is to determine when the classes are consistent with a sequence, and if they are, to find such a sequence. These papers show that the problem can be solved in polynomial time.

The second group of papers analyze approximation algorithms for the shortest common superstring problem, which as we have indicated is equivalent to the sequence reconstruction problem without error, and with fragment orientation known. Tarhio and Ukkonen [36], Turner [38], and Ukkonen [39] show that a simple greedy algorithm finds a superstring whose amount of compression is within a factor of one-half of the maximum, and give efficient implementations. Blum *et al.* [1] prove that the greedy algorithm delivers a superstring at most four times longer than the shortest, and that a simple variant delivers a superstring at most three times longer than the shortest. It is not known whether these bounds are tight. Li [21] examines sequence assembly from the viewpoint of computational learning theory, and shows that an approximation algorithm for SUPERSTRING will learn the underlying sequence in polynomial time in the PAC model of learning, given fragments without error and with known orientation.

In the third group of papers, Staden [35], Gingeras *et al.* [14], and Peltola *et al.* [27] develop software for sequence assembly. Peltola *et al.* [26] describe the algorithms used in [27], and also give the first statement of the sequence reconstruction problem. These papers deal with error, and with orientation, but do not characterize the quality of the output.

In addition, three papers have recently come to our attention that examine the subtask of computing overlaps between pairs of fragments. Gusfield *et al.* [15] show that with the suffix tree data structure, the longest overlap between a suffix of one fragment and a prefix of another can be determined for all pairs of fragments in time linear in the size of the input and output, if no errors are permitted in the overlaps. Cull and Holloway [6] apply the suffix array data structure of Manber and Myers [22] to find overlaps, where fragments are assumed to contain only substitution errors, and the suffix and prefix of each fragment is assumed to match only one other fragment in an overlap longer than a given threshold. Huang [16] applies a local alignment algorithm of Smith and Waterman [34] to compute an overlap that maximizes a linear function of the number of exact matches and errors in the alignment, and uses a filtering technique of Chang and Lawler [4] to avoid considering some of the pairs of fragments whose alignment score is below a fixed threshold.

Our work may be distinguished from prior theoretical investigations in that we address both sequencing errors and unknown orientation, and in contrast to prior software, each phase is a well-defined problem. Like Peltola *et al.* [26], we cannot claim that our algorithm as a whole solves RECONSTRUCT, but in distinction each phase solves or approximates a precise optimization problem. Moreover, for the case of no error and known orientation, we can say that our algorithm without modification solves Reconstruct, which is equivalent to Superstring. In this sense, the algorithm generalizes earlier theoretical work.

Overview. Our algorithm proceeds in four phases consisting of the following combinatorial problems:

1. Constructing a graph of approximate overlaps between pairs of fragments.
2. Assigning an orientation to the fragments, i.e., choosing the forward or reverse complement sequence for each fragment.
3. Selecting a set of overlaps that induce a consistent layout of the oriented fragments.
4. Merging the selected overlaps into a multiple sequence alignment, and voting on a consensus.

We devote a section of the paper to each of the four phases.

In phase 1, we compute overlaps within the error rate that maximize a likelihood function on alignments. Edges in an *overlap graph* correspond to these alignments, and are weighted by likelihood. Given fragments of total length N and error rate ε, our method for computing the graph modeling these overlaps takes $O(\varepsilon N^2)$ time.

In phase 2, we orient fragments so as to maximize the weight of all edges in the overlap graph that are consistent with the chosen orientation. This subproblem is NP-complete. We present an exact algorithm that computes an optimal orientation in $O(K(EV))$ time for an overlap graph of V fragments and E edges, where $K \leq 2^V$ is the size of its branch-and-bound search tree. We also present an approximation algorithm that computes an orientation of weight at least one-half the maximum in $O(E + V \log V)$ time.

In phase 3, we place the fragments in an overlapping layout by selecting a set of edges of maximum total weight that form a branching satisfying a *dovetail-chain* property. Finding such a branching is also NP-complete. We present an exact algorithm that computes an optimal layout by finding a maximum weight dovetail-chain branching in $O(K(E + V \log V))$ time, where $K \leq 2^E$ is the size of its search tree. A greedy approximation algorithm for this problem is well known, and in contrast finds a branching of weight at least one-third the maximum in $O(E \log V)$ time. We further show that our approach naturally lends itself to producing alternate layouts, if desired.

In phase 4, we take the set of all overlaps in the graph that agree with the fragment layout, and merge them into a multiple sequence alignment as follows. The alignments represented by the set of overlaps match pairs of characters from the fragments. Of these character pairs, we seek a subset of maximum total weight that forms a multiple alignment. This problem is also NP-complete, though it can be solved in time exponential in the maximum number of fragments that mutually overlap in the layout. Given overlaps that match M pairs of characters from a layout with at most D mutually overlapping fragments, we construct a multiple sequence alignment of length L, and a consensus sequence, in $O(D^2L + M + N)$ time. The set of matched pairs that forms the alignment has weight at least $2/D$ of the maximum.

The paper closes with a presentation of some preliminary experimental results for the combined algorithm, and we conclude by suggesting some possible extensions.

2. Overlap Graph Construction.

Our algorithm forms a reconstruction by overlapping the fragments in pairs. We represent the set of all pairwise overlaps with a directed edge-weighted graph called an *overlap graph*. This section describes the structure of this graph, and how it is constructed.

If we think of fragments as intervals, and overlaps as intersections of intervals, there are essentially four ways a pair of fragments can overlap, as shown in Figure 1. Each type of overlap is an alignment between the sequence for fragment A and the sequence for fragment B. If the alignment is between a proper suffix[4] of A and a proper prefix of B, we call it a *dovetail*, and say *A dovetails to B*. If the alignment is between a substring of A and all of B, we call it a *containment*, and say *A contains B*. An *overlap* of A and B is denoted by an ordered pair (A, B), and represents an alignment where either A dovetails to B, or A contains B. The overlap is *at rate* ε if the number of errors in the alignment is at most $\varepsilon|A| + \varepsilon|B|$, where an error is the insertion, deletion, or substitution of a character. We also attribute an overlap with a real-valued weight, which is a score for the alignment based on the probability of the overlap occurring by chance. There are many possible alignments for a given type of overlap; we choose an alignment that has maximum score.

An overlap graph $G = (V, E, w)$ represents fragments with vertex set V, and overlaps with edge set E. Edge weight function w gives the weight of an overlap. In an *unoriented overlap graph*, V contains two vertices for every fragment F. One vertex represents sequence F, and the other represents the reverse complement sequence $\overleftarrow{F}$. An overlap (A, B) of sequence A with sequence B is represented by an edge directed from vertex A to vertex B. An edge corresponding to a dovetail is denoted by $A \rightarrow B$; a containment is denoted by $A \Rightarrow B$.

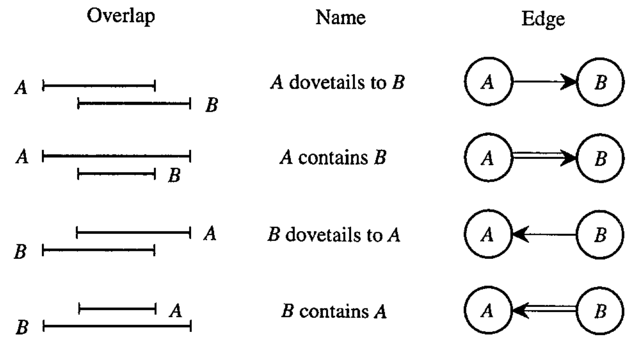

Fig. 1. The four types of overlaps.

[4] A *proper* suffix of $a_1 a_2 \cdots a_n$ is a substring $a_i a_{i+1} \cdots a_n$ where $i > 1$. A proper prefix is a substring $a_1 a_2 \cdots a_i$ where $i < n$.

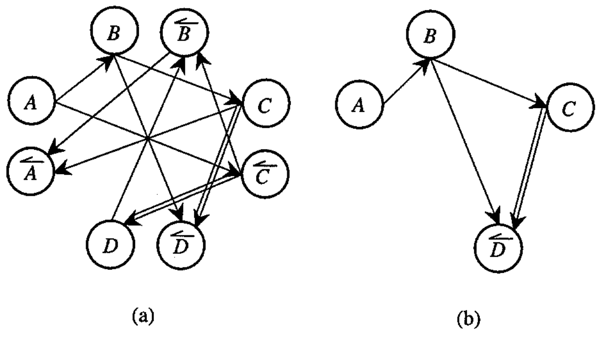

Fig. 2. Overlap graphs. (a) unoriented graph. (b) An oriented subgraph.

Note that edge $\overleftarrow{B} \rightarrow \overleftarrow{A}$ is equivalent to edge $A \rightarrow B$. In other words, any alignment between a suffix of $\overleftarrow{B}$ and a prefix of $\overleftarrow{A}$ is an alignment between a suffix of A and a prefix of B. Similarly, edge $\overleftarrow{A} \Rightarrow \overleftarrow{B}$ is equivalent to edge $A \Rightarrow B$. This equivalence is reflected in our representation: $A \rightarrow B$ is always accompanied by $\overleftarrow{B} \rightarrow \overleftarrow{A}$, and $A \Rightarrow B$ is always accompanied by $\overleftarrow{A} \Rightarrow \overleftarrow{B}$. With this pairing of edges, there are essentially four possible overlaps between two fragments when their orientation is unknown, namely, (A, B), (B, A), $(A, \overleftarrow{B})$, and $(\overleftarrow{B}, A)$. Each of these overlaps may be a dovetail or a containment.

Our algorithm for overlap graph construction builds an unoriented graph. We form an *oriented overlap graph* from an unoriented graph G by specifying the orientation of fragments. In this case, we restrict our attention to the subgraph of G induced by the vertices specified in the orientation.[5] For example, Figure 2 shows an unoriented overlap graph, and the subgraph induced by a particular orientation. Section 3 describes how we determine which oriented subgraph to submit to the fragment layout algorithm of Section 4.

2.1. The Least-Random Overlap Problem. Given two fragments, we would like to infer how they overlap in the underlying sequence, if they overlap at all. We model this inference problem as one of finding, for each type of overlap, an alignment of minimum probability. If this alignment is statistically rare, it is unlikely to be due to a chance matching of characters. An overlap poorly explained by chance is likely to represent a true overlap of the fragments.[6]

To determine the probability of an alignment, we treat the fragments as random sequences with each character drawn uniformly and independently from the alphabet $\{\mathsf{a}, \mathsf{c}, \mathsf{g}, \mathsf{t}\}$. While the exact probability of an alignment is unknown even for this model, a result of Chvátal and Sankoff [5] on random common subsequences gives a good upper bound.

[5] The subgraph of (V, E) *induced by* a subset $V' \subseteq V$ is the graph (V', E') where $E' \subseteq E$ contains only edges joining vertices in V'.

[6] The other possibility is that the sequence contains a repeat.

The alignments that we compute match a pair of characters only when they are equal. These matches yield a common subsequence of the fragments, and each unmatched character is considered an insertion or a deletion error. (Thus a substitution is counted as a deletion followed by an insertion.) The quantities that we measure for an alignment are l, the length of the common subsequence, and d, the number of errors. Sankoff and Chvátal [30] show that the number of sequences of length $l + d$ over an alphabet of size s that contain a fixed subsequence of length l is

$$N_s(l, d) = \sum_{0 \le i \le d} \binom{l+d}{i} (s-1)^i, \tag{1}$$

independent of the particular subsequence. This gives an upper bound on the probability of an alignment with l matches and d errors of

$$P_s(l, d) = \sum_{0 \le i \le d} N_s(l, i)\, N_s(l, d-i) \Big/ (d+1) s^{l+d}. \tag{2}$$

We say an overlap minimizing $P_s(l, d)$ is *least random*. Our problem is the following.

DEFINITION. The *least-random overlap problem* (OVERLAP) is, given an ordered pair A, B of sequences from an alphabet of size s and an error rate ε, find an overlap (A, B) at rate ε minimizing $P_s(l, d)$, where l is the number of exact matches in the alignment and d is the number of errors.

We make a few remarks on the problem. Our sequences are from an alphabet of size four, so we are interested in minimizing $P_4(l, d)$. Minimizing $P_4(l, d)$ is equivalent to maximizing

$$L(l, d) = -\log_4 P_4(l, d),$$

which we call the *likelihood* of the alignment. Likelihood has several nice properties. $L(l, d)$ is decreasing in l, decreasing in d, and bounded by l. While we do not prove it here, asymptotically $L(l, d) = O(l - d \log(l/d))$. Likelihood balances matches against error in an objective manner.[7]

[7] Intuitively we would like an overlap with the greatest number of matches, yet matches are often achieved at the cost of error in the alignment. The packages of Staden [35] and Gingeras *et al.* [14] use rules of thumb such as, extend an alignment with five matches if this can be done with only three errors, while the system of Peltola *et al.* [26] tries to minimize d/l, which has the rough behavior of $L(l, d)$, but does not discriminate between longer and shorter overlaps with the same error density. Huang [16] minimizes $l - d$, which also approximates $L(l, d)$, but trades matches against errors linearly, which from an objective point of view is arbitrary. Peltola *et al.* [26] and Huang [16], however, are able to accommodate substitution errors within their objective function.

2.2. Computing Overlaps. A brute-force algorithm for OVERLAP would be to:

1. Compute, for all possible dovetails and containments between A and B, the edit distance between the overlapped substrings.
2. Evaluate $L(l, d)$ for each of these overlaps, where $d \leq \varepsilon|A| + \varepsilon|B|$ is the distance between the overlapped substrings $\tilde{A}$ and $\tilde{B}$, and $l = \frac{1}{2}(|\tilde{A}| + |\tilde{B}| - d)$.
3. Output the overlap of maximum $L(l, d)$.

In step 1 of this algorithm, it suffices to consider an alignment of minimum distance for each possible overlap, because $L(l, d)$ is monotone in its arguments. Since a pair A, B of sequences of length m and n has $O(mn)$ dovetails and $O(m^2)$ containments, and since the edit distance for each dovetail and containment may be computed in $O(mn)$ time by the standard dynamic programming algorithm [31], this gives an $O(m^2n^2)$-time algorithm—and it is easy to bring this down to $O(m^2n)$ time by combining subproblems. (We assume here that $m \leq n$.)

Myers [25] has shown, however, that it is possible to solve all $O(mn)$ subproblems in $O(\delta n)$ time, where δ is the maximum edit distance allowed.[8] In our application, $\delta = \lfloor \varepsilon m + \varepsilon n \rfloor$, so this gives an $O(\varepsilon n^2)$-time algorithm.

The key idea of Myers's algorithm is to solve the alignment problems incrementally, and to represent the successive solutions with a data structure that can be efficiently updated. For $S = s_1 s_2 \cdots s_n$, let $S_{i,j}$ denote the substring $s_i s_{i+1} \cdots s_j$, let $S_{i,*}$ denote the suffix $S_{i,n}$, and let $S_{*,j}$ denote the prefix $S_{1,j}$. Given sequences A and B of length n and m, Myers solves a series of n alignment problems that compare increasingly longer suffixes $A_{k,*}$ against B. For each suffix $A_{k,*}$, the edit distance is obtained between all $A_{k,i}$ and all $B_{*,j}$ for which $d(A_{k,i}, B_{*,j}) \leq \delta$. Note this includes the dovetails $(A_{k,*}, B_{*,j})$ and the containments $(A_{k,i}, B_{*,*})$. These distances are not explicitly computed, but are represented implicitly by a sparse data structure that encodes their values. Any particular distance, if needed, can be recovered from the encoding.

Given the encoding for $A_{k+1,*}$ versus B, Myers shows that the encoding for $A_{k,*}$ versus B can be obtained in $O(\delta)$ time. To find a least random overlap, the distances we need, for a fixed k, are $d(A_{k,*}, B_{*,j})$ for the m possible dovetails, and $d(A_{k,i}, B_{*,*})$ for the roughly n possible containments. Of these $O(m + n)$ dovetails and containments, only $O(\delta)$ can have distance at most δ. With the encoding for $A_{k,*}$ versus B in hand, these $O(\delta)$ distances can be recovered in $O(\delta)$ time. Given the distances, we can evaluate the likelihoods. This spends a total of $O(\delta)$ time per problem, and as there are n problems, it gives an $O(\delta n)$-time algorithm.

Throughout this description we have assumed that the likelihood function can be evaluated in $O(1)$ time. Computing $L(l, d)$ directly from (1) and (2), however, involves a sum of $O(d^2)$ terms. It is possible, however, to precompute a table of $L(l, d)$, since l and d are both bounded in practice. For fragemnts of at most 1000 nucleotides, and error rates of at most 10%, it suffices to store a table of $L(l, d)$ for $0 \leq l \leq 1000$ and $0 \leq d \leq 200$.

[8] This is possible because only $O(\delta n)$ subproblem can have distance at most δ. Nevertheless, it is remarkable that a subproblem can be solved in effectively constant time when in isolation it takes $O(\delta n)$ time. Myers does assume edit distance is measured in terms of insertions and deletions only.

To construct an overlap graph then for fragments at error rate ε, for every pair A, B of fragments we solve OVERLAP for (A, B), (B, A), $(A, \overleftarrow{B})$, and $(\overleftarrow{B}, A)$, using Myers's algorithm. For fragments of total length N, this takes $O(\varepsilon N^2)$ time. Each overlap is classified as a dovetail or a containment, and we add the appropriate edge to the graph attributed with the corresponding alignment, and weighted by the likelihood of the match. Alignments are encoded by edit scripts to conserve space (see Section 5.3.4).

2.3. Culling Overlaps. The construction we have described gives a complete overlap graph. Most of the edges, however, will represent chance alignments rather than true overlaps. We now describe how to cull such edges from the graph. In practice we observe that culling reduces the number of edges from $O(V^2)$ to $O(V)$. Our orientation and layout algorithms take advantage of this sparsity.

2.3.1. Match Significance. We use two criteria for culling edges, the first based on match probability. We assume the biologist has a *match significance threshold* λ for the minimum acceptable likelihood of an overlap. An overlap of l matches and d errors is rejected if

$$L(l, d) < \lambda.$$

With edges weighted by likelihood, this implies that every edge in the graph has weight at least λ. Since $L(l, d) \leq l$, it also implies that every overlap must have at least $\lceil \lambda \rceil$ matches. Table 1 lists the minimum number of matches to achieve a given threshold, for various values of λ and d.

2.3.2. Error Distribution. Our second criterion for culling edges is based on the distribution of errors in an overlap. The alignment for an edge between fragments

Table 1. Number of matches to achieve $L(l, d) \geq \lambda$.

	Matches, l			
Errors, d	$\lambda = 5$	$\lambda = 10$	$\lambda = 25$	$\lambda = 50$
0	5	10	25	50
1	7	12	28	53
2	8	14	30	56
3	9	15	32	58
4	10	17	33	60
5	12	18	35	62
10	18	25	44	72
15	24	31	51	81
20	30	38	59	89
25	37	45	66	97
50	69	78	101	136
75	102	111	135	172
100	135	144	169	207

A and B is guaranteed to have at most $\varepsilon|A| + \varepsilon|B|$ errors, but an overlap that aligns substrings $\tilde{A}$ and $\tilde{B}$ is expected to have around $\varepsilon|\tilde{A}| + \varepsilon|\tilde{B}|$ errors. If the number of errors far exceeds this, it is natural to suspect that the edge is not a true overlap, and reject it. Such an edge is inconsistent with the hypothesis that errors are roughly evenly distributed.[9]

Let us assume that fragments A and B have a total of $n = \lfloor \varepsilon|A| + \varepsilon|B| \rfloor$ errors between them, and that the probability of observing an error in an overlap of substrings $\tilde{A}$ and $\tilde{B}$ is $p = (|\tilde{A}| + |\tilde{B}|)/(|A| + |B|)$. Then, if errors are independent, the number of errors in the overlap, D, is a binomial random variable with parameters n and p. The probability of observing d or more errors is

$$\Pr\{D \geq d\} = \sum_{d \leq i \leq n} \binom{n}{i} p^i (1-p)^{m-i}.$$

To cull overlaps on the basis of error distribution, we assume that the biologist is willing to reject a small fraction of all alignments: those that do not distribute the n errors evenly, but have an error count d exceeding a critical value. We call this fraction the *error distribution threshold* ξ, and reject an overlap with d errors if

$$\Pr\{D \geq d\} \leq \xi. \tag{3}$$

This is illustrated in Figure 3. The probability $Pr\{D \geq d\}$ is equal to $I_p(d, n-d+1)$, where $I_x(a, b)$ is the incomplete beta function. There are fast numerical methods for evaluating $I_x(a, b)$, which yield an efficient test of inequality (3). (See, for instance, pp. 178–180 of [28].)

Finally, we note that both the match significance and the error distribution criteria are needed. Without a match significance criterion, time and space are wasted on short overlaps, such as those that align one character. Without an error distribution criterion, long but poor overlaps are permitted, such as those that align many characters, but have an error rate of 50%. Also note that the extreme case $\lambda = 0$ and $\xi = 0$ is permitted, in which case no overlaps are rejected.

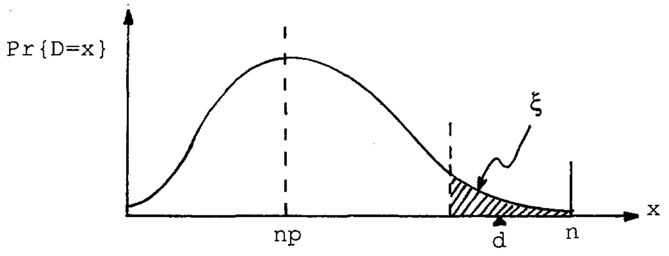

Fig. 3. Culling an overlap by error distribution.

[9] Admittedly, when the fragment sequences are obtained from reading electrophoresis gels, errors occur more frequently at the fragment ends. We can conservatively treat such errors as evenly distributed according to the maximum error rate at the end of a fragment.

To summarize, given fragments of total length N, error rate ε, a match signficance threshold, and an error distribution threshold, the first phase of our algorithm constructs a graph of least-random overlaps, weighted by likelihood, in time $O(\varepsilon N^2)$.

3. Fragment Orientation.

Once we have constructed an overlap graph G and culled its edges, we are left with a collection of significant overlaps. With high probability these edges represent true overlaps between fragments, and while some may align reverse complement sequences and others not, the majority of the overlaps should indicate a consistent orientation of the fragments. This section describes how we find an oriented subgraph of G in preparation for fragment layout.

3.1. The Fragment Orientation Problem. To specify an oriented subgraph is to determine, for every fragment F, whether sequence F or $\overleftarrow{F}$ is used in the reconstruction. If we decide to use F, we say the fragment is assigned the *forward orientation*; if we use $\overleftarrow{F}$, we say it is assigned the *reverse orientation.* Assigning orientations eliminates some overlaps and retains the possibility of using others. For example, assigning A and B the forward orientation estimates any overlap between A and $\overleftarrow{B}$, but allows us to use (A, B) *or* (B, A).

For A and B from an overlap graph G with edge weight function w, let

$$\text{same}(A, B) = \max(w(A, B), w(B, A))$$

and

$$\text{opp}(A, B) = \max(w(w(A, \overleftarrow{B}), w(\overleftarrow{B}, A)),$$

with the understanding that the weight of an overlap is zero when it is not in G. The weight of the best overlap between fragments A and B is given by same(A, B) when both are assigned the same orientation; opp(A, B) applies when they are assigned opposite orientations. Note that same(A, B) and opp(A, B) are symmetric in their arguments. We view these functions as definining an undirected graph $\tilde{G}$ whose vertices are the fragments and whose edges are the pairs $\{A, B\}$ for which same(A, B) or opp(A, B) is nonzero. With this interpretation, same and opp are two edge-weight functions for $\tilde{G}$. For an overlap graph with V fragments and E overlaps, $\tilde{G}$ has V vertices and at most E edges.

An orientation of a collection $\mathcal{F}$ of fragments is represented by a partition $(\mathcal{O}, \mathcal{F} - \mathcal{O})$ where $\mathcal{O} \subseteq \mathcal{F}$ is the set of fragments in the forward orientation, and $\mathcal{F} - \mathcal{O}$ is the set with the reverse orientation. We write $\bar{\mathcal{O}}$ for $\mathcal{F} - \mathcal{O}$ when $\mathcal{F}$ is given by context, and often specify partition $(\mathcal{O}, \bar{\mathcal{O}})$ by only giving set $\mathcal{O}$. For an edge $\{A, B\}$ of $\tilde{G}$ and a partition $(\mathcal{O}, \bar{\mathcal{O}})$ of $\mathcal{F}$, we use the notation $(A, B) \in (\mathcal{O}, \bar{\mathcal{O}})$ to indicate that $A \in \mathcal{O}$ and $B \in \bar{\mathcal{O}}$, and we write $(A, B) \notin (\mathcal{O}, \bar{\mathcal{O}})$ when $A, B \in \mathcal{O}$ or $A, B \in \bar{\mathcal{O}}$.

Since all overlaps remaining after culling are significant, we would like an

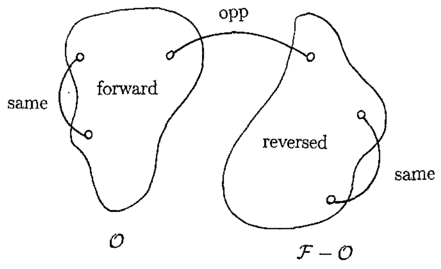

Fig. 4. The weight of an orientation $\mathcal{O}$.

orientation $(\mathcal{O}, \bar{\mathcal{O}})$ that minimizes the weight of the overlaps it eliminates, or equivalently maximizes the weight of the overlaps it retains. Our problem is the following.

Definition. The *fragment orientation problem* (Orient) is, given fragments $\mathcal{F}$ and functions same and opp, find an orientation $\mathcal{O}$ for $\mathcal{F}$ maximizing

$$w(\mathcal{O}) = \sum_{(A,B)\in(\mathcal{O},\bar{\mathcal{O}})} \mathrm{opp}(A, B) + \sum_{(A,B)\notin(\mathcal{O},\bar{\mathcal{O}})} \mathrm{same}(A, B).$$

We call $w(\mathcal{O})$ the weight of the orientation. Figure 4 gives an illustration.

Orient is NP-complete. It is polynomial-time equivalent to the maximum-weight cut problem [18].

3.2. *An Approximation Algorithm.*

While finding an optimal orientation is hard, it is easy to find an orientation that is close to optimal.

Given an ordering $F_1, F_2, \ldots, F_n$ of $\mathcal{F}$ we compute an orientation $\mathcal{O}$ as follows. Initially, we set $\mathcal{O} := \{\ \}$. At step i, we consider adding F_i to $\mathcal{O}$ where $(\mathcal{O}, \bar{\mathcal{O}})$ currently partitions $\{F_1, \ldots, F_{i-1}\}$. If $w(\mathcal{O} \cup \{F_i\}) \geq w(\bar{\mathcal{O}} \cup \{F_i\})$, we set $\mathcal{O} := \mathcal{O} \cup \{F_i\}$. Otherwise, we leave $\mathcal{O}$ unchanged, which effectively adds F_i to $\bar{\mathcal{O}}$. After n steps, we output $\mathcal{O}$.

Each decision of this *greedy algorithm* involves only the edges incident to the current fragment F_i. Given an ordering of $\mathcal{F}$, it runs in $O(E + V)$ time and $O(V)$ space for a graph of V vertices and E edges.

The greedy algorithm guarantees an orientation of weight at least $\frac{1}{2}w(\mathcal{O}^*)$, where $\mathcal{O}^*$ is an optimal orientation. (The following analysis is the same as that of a folklore heuristic for maximum weight cut.) To see this, note that a trivial upper bound on $w(\mathcal{O}^*)$ is the total weight of the graph,

$$\sum_i \sum_{j<i} (\mathrm{same}(F_i, F_j) + \mathrm{opp}(F_i, F_j)).$$

If the greedy algorithm adds F_i to $\mathcal{O}$ at step i, the weight of the orientation increases by

$$\sum_{j<i}\left(\sum_{F_j\in\mathcal{O}}\operatorname{same}(F_i, F_j)+\sum_{F_j\in\bar{\mathcal{O}}}\operatorname{opp}(F_i, F_j)\right). \tag{4}$$

If it does not add F_i to $\mathcal{O}$, the weight increases by

$$\sum_{j<i}\left(\sum_{F_j\in\mathcal{O}}\operatorname{opp}(F_i, F_j)+\sum_{F_j\in\bar{\mathcal{O}}}\operatorname{same}(F_i, F_j)\right). \tag{5}$$

Denote the actual amount of increase at step i by $\Delta w(\mathcal{O}_i)$. Since the greedy algorithm chooses the greater of (4) and (5),

$$2\Delta w(\mathcal{O}_i) \geq \sum_{j<i}(\operatorname{same}(F_i, F_j)+\operatorname{opp}(F_i, F_j)).$$

Summing over all steps,

$$w(\mathcal{O})=\sum_i \Delta w(\mathcal{O}_i) \geq \tfrac{1}{2}\sum_i\sum_{j<i}(\operatorname{same}(F_i, F_j)+\operatorname{opp}(F_i, F_j)) \geq \tfrac{1}{2}w(\mathcal{O}^*).$$

This is a worst-case bound, and it holds for any order of the fragments. In a good order, the difference in weight between being in and out of $\mathcal{O}$ should be large for fragments occurring early in the order. Otherwise, unrelated fragments that occur early in the order may receive an arbitrary orientation.

We can determine such an order as follows. Given $\mathcal{F}$, same, and opp, we form an undirected graph with vertex set $\mathcal{F}$ and edge weight function

$$w(A, B) = |\operatorname{same}(A, B) - \operatorname{opp}(A, B)|.$$

Over this graph, we compute a maximum-weight spanning tree T. Such a tree clusters vertices by edge weight. We then select a root R of greatest total distance from all other vertices, where the distance between vertices A and B is the number of edges on the path connecting them in T. As T tends toward a single path, R tends toward an endpoint of the path. Finally, we order the fragments depth-first in T from R. The intuition behind this heuristic is that we expect the graph on which we compute a spanning tree to be the interval graph given by the correct layout of the fragments, with some weak edges thrown in. For such a graph, the fragment order given by the heuristic will tend toward the fragment order given by the underlying layout.

Constructing the graph takes $O(E + V)$ time, where V is the number of fragments and E is the number of pairs of fragments with nonzero same or opp. Tree T can be found in $O(E + V \log V)$ time [9]. We can locate R in $O(V)$ time by two passes over T. The first pass computes the total distance of each vertex A to all vertices in the subtree rooted at A, bottom-up, along with the size of the subtree at A. The second pass uses this information to compute the total distance of each vertex in the whole tree, top-down, while keeping track of the vertex of maximum total distance.

Determining the fragment order then takes $O(E + V \log V)$ time, and $O(E + V)$ space, which dominates the time and space for the greedy algorithm.

3.3. An Exact Algorithm. Using the idea of processing fragments in order, we can also design an *exact algorithm* that computes an optimal orientation. Given an ordering $F_1, \ldots, F_n$ of fragments $\mathscr{F}$, let $\mathscr{F}_i^k$ denote the subset $\{F_i, F_{i+1}, \ldots, F_k\}$, and let $\mathscr{F}^k$ denote $\mathscr{F}_1^k$. We compute an optimal orientation for $\mathscr{F}^1, \mathscr{F}^2, \ldots, \mathscr{F}^n$, using the solutions for smaller problems to solve larger ones.

Each problem $\mathscr{F}^k$ is solved using the branch-and-bound technique. The computation can be viewed as a binary tree of height k, as shown in Figure 5. A node at height i assigns an orientation to F_i, and a root-to-leaf path assigns an orientation to all fragments in $\mathscr{F}^k$. We can arbitrarily assign the forward orientation to F_k, since pairs of solutions with opposite orientations are equivalent.

As the exact algorithm explores the tree from the root to a node of height i, it accumulates an orientation $\mathscr{O}_i^k$ of $\mathscr{F}_i^k$. On describing to height $i - 1$, it extends this orientation, first by adding F_{i-1} to $\mathscr{O}_i^k$, which takes the left branch in the tree, and later returning to add F_{i-1} to $\bar{\mathscr{O}}_i^k$, which takes the right branch.

When considering a move, branches are eliminated using subproblems that have already been solved. When tackling $\mathscr{F}^k$ for instance, the solutions to $\mathscr{F}^1, \ldots, \mathscr{F}^{k-1}$ are in hand. This allows us to compute quickly an upper bound on the weight $w(\mathscr{O})$ of any completion of $\mathscr{O}_i^k$ to a partition $(\mathscr{O}, \bar{\mathscr{O}})$ of $\mathscr{F}^k$.

For subsets $\mathscr{A}$ and $\mathscr{B}$ of $\mathscr{F}^k$, let us write $w(\mathscr{O})|_{\mathscr{A},\mathscr{B}}$ for

$$\sum_{\substack{A \in \mathscr{A} \\ B \in \mathscr{B} \\ (A,B) \in (\mathscr{O}, \bar{\mathscr{O}})}} \operatorname{opp}(A, B) + \sum_{\substack{A \in \mathscr{A} \\ B \in \mathscr{B} \\ (A,B) \notin (\mathscr{O}, \bar{\mathscr{O}})}} \operatorname{same}(A, B),$$

the weight of orientation $\mathscr{O}$ restricted to edges that have one endpoint in $\mathscr{A}$ and

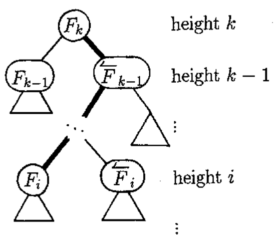

Fig. 5. The branch-and-bound search tree for problem $\mathscr{F}^k$. A root-to-leaf path assigns an orientation to $F_k, F_{k-1}, \ldots, F_1$.

one endpoint in $\mathcal{B}$. For any partition $(\mathcal{A}, \mathcal{B})$ of $\mathcal{F}^k$,

$$w(\mathcal{O}) = w(\mathcal{O})|_{\mathcal{A},\mathcal{A}} + w(\mathcal{O})|_{\mathcal{B},\mathcal{B}} + w(\mathcal{O})|_{\mathcal{A},\mathcal{B}}. \tag{6}$$

Let $\mathcal{A} = \mathcal{F}^{i-1}$ and $\mathcal{B} = \mathcal{F}_i^k$. For this choice of $\mathcal{A}$ and $\mathcal{B}$, we can upper bound each term of (6), which represents the weight of an extension of $\mathcal{O}_i^k$ to the fragments in $\mathcal{A} = \mathcal{F}^{i-1}$, as follows. The first term, $w(\mathcal{O})|_{\mathcal{A},\mathcal{A}}$, is at most the weight of an optimal orientation for $\mathcal{F}^{i-1}$, which is known. The second term, $w(\mathcal{O})|_{\mathcal{B},\mathcal{B}}$, is exactly $w(\mathcal{O}_i^k)$, which is the weight of the partial orientation. The third term, $w(\mathcal{O})|_{\mathcal{A},\mathcal{B}}$, is at most

$$\sum_{A \in \mathcal{A}} \max(w(\mathcal{O}_i^k \cup \{A\})|_{\{A\},\mathcal{B}}, w(\bar{\mathcal{O}}_i^k \cup \{A\})|_{\{A\},\mathcal{B}}).$$

This upper bound for $w(\mathcal{O})|_{\mathcal{A},\mathcal{B}}$ allows the fragments of $\mathcal{A} = \mathcal{F}^{i-1}$ to join $\mathcal{O}_i^k$ or $\bar{\mathcal{O}}_i^k$ optimistically, independent of how they are partitioned in the first term.

The sum of the bounds for these three terms is an upper bound on $w(\mathcal{O})$, the best extension of $\mathcal{O}_i^k$ to $\mathcal{F}^k$. Call this upper bound U_i^k. We can also maintain a lower bound L^k on the weight of a solution to $\mathcal{F}^k$. Initially, L^k may be obtained by greedily adding F_k to the solution for $\mathcal{F}^{k-1}$. Whenever our search reaches a leaf corresponding to an orientation of greater weight, we raise L^k. If we discover at height i that $U_i^k \le L^k$, it is not worth searching the subtree further, and we backtrack. In this way the exact algorithm avoids exploring the whole search tree.

Evaluating the upper bound at a node takes $O(E + V)$ time. The bounds for the first and second terms of (6) can be looked up in constant time if the weights of solutions to subproblems are saved, and the weight of $\mathcal{O}_i^k$ is accumulated while descending the tree. Bounding the third term involves looking at no more than E edges and V vertices. If K nodes are explored in the search trees for $\mathcal{F}^1$ through $\mathcal{F}^n$, the total time is $O(K(E + V))$, where $K \le 2^V$.

The space for subproblems is $O(V)$, as only the weights of solutions need to be stored,[10] and the stack to transverse a tree has height at most V. The total space then is $O(E + V)$.

A feature of the exact and the approximation algorithm is that they lend themselves to a *hybrid algorithm* that enjoys some of the advantages of both. Suppose the biologist places a bound on the maximum number of nodes to explore in any search tree. When we run the exact algorithm on problems $\mathcal{F}^1, \mathcal{F}^2, \ldots,$ we count the number of nodes explored in the current tree. If on problem $\mathcal{F}^k$ this count exceeds the bound, we stop the exact algorithm, take the optimal orientation for problem $\mathcal{F}^{k-1}$, and extend it with the greedy algorithm to an orientation of $\mathcal{F}^n$.

The argument of the previous section shows that this hybrid algorithm achieves at least the factor of one-half attained by the greedy algorithm. Moreover, it has the capacity to compute an optimal orientation (and prove that it has found one) while always running in polynomial time. Though we have solved problems of

[10] Note that storing the orientations for the solutions to $\mathcal{F}^1, \ldots, \mathcal{F}^n$ would take $O(V^2)$ space.

500 fragments with the exact algorithm, some instances of 250 fragments have proven difficult to solve to optimality. With a bound of say 1000 nodes, the transition from the exact algorithm to the approximation algorithm can be made at run time.

Finally, we note that all of these algorithms can accommodate orientation constraints. For example, the user may know that some fragments should be forward and others reversed, or that some pairs of fragments should have the same orientation, while others should have opposite orientations. Since all of these algorithms form an orientation one fragment at a time, any constraint that applies to a fragment can be checked when making an orientation decision. Thus, constraints can be accommodated in time linear in the number of constrained fragments and constrained pairs.

4. Fragment Layout.

The fragment orientation computed by the second phase of our algorithm induces an oriented subgraph of the overlap graph. In the third phase, we select edges from this subgraph that are consistent with an interpretation of fragments as intervals of the line. These intervals represent substrings of the underlying sequence, and we call the ensemble of fragment intervals a *fragment layout*. Section 4.1 describes the structure of a set of edges in an overlap graph that corresponds to a layout. We call such a set a *dovetail-chain branching*. Sections 4.2–4.5 describe our algorithm for computing an optimal dovetail-chain branching, and Section 4.6 describes how to compute alternate branchings for a user who desires alternate layouts.

4.1. *The Dovetail-Chain Branching Problem.*

Consider a sequence S that is a reconstruction for fragments $\mathscr{F}$. Every fragment $F \in \mathscr{F}$ matches a substring of S, say the substring from the ith character to the jth character of S. Through this substring, reconstruction S associates the interval $[i, j]$ with fragment F. We call the collection of intervals for the fragments a *fragment layout* for $\mathscr{F}$.

For a layout $\mathscr{L}$, let $\mathscr{L}(F)$ denote the interval for fragment F. The *length* of $\mathscr{L}$, denoted by $|\mathscr{L}|$, is $|\bigcup_{F \in \mathscr{F}} \mathscr{L}(F)|$, the length of the total interval covered by $\mathscr{L}$. Clearly, a shortest reconstruction will have the shortest associated layout.

With every fragment layout we can also associate a set of edges in an overlap graph. For fragments whose intervals are identical, we form an equivalence class, select one representative fragment for the class, and direct a containment edge from the representative to every other fragment in its class. Now remove any fragment that is not a representative from consideration. The remaining fragments can be totally ordered, first by increasing left endpoint, and second by decreasing right endpoint. For any fragment F whose interval is contained in another's, there is a least interval in the order that contains F. We direct a containment edge from this least fragment to F. Now remove any fragment whose interval is contained in another's. The remaining fragments have increasing left and right endpoints. Direct a dovetail edge from a fragment to its successor in the order if their intervals overlap. The resulting set of edges satisfies four properties:

1. Every vertex has at most one incoming edge.
2. The edges do not form cycles.
3. No two dovetail edges leave the same vertex.
4. No containment edge $A \Rightarrow B$ is followed by a dovetail edge $B \rightarrow C$.

A set $\mathscr{B}$ of edges that satisfies properties 1 and 2 is called a *branching.* A branching may also be characterized as a collection of vertex-disjoint trees with edges directed outward from the *roots*, which are the vertices with no in-edge. Each directed tree of the branching is called an *arborescence.* A set of edges that, in addition, satisfies properties 3 and 4, we call a *dovetail-chain branching.* Its dovetail edges form disjoint chains that originate at the roots.

Following Staden [35], we call a maximal set of fragments that cover a contiguous interval in the layout, *a contig.* Note that the contigs of the layout correspond to the arborescences of the branching.

Just as we can associate with every layout a dovetail-chain branching, with every branching we can also associate a fragment layout. For an overlap (A, B), let indent(A, B) be the length of the prefix of A that is not aligned to B. Given a branching $\mathscr{B}$ of overlaps, we can construct a contig for each arborescence of $\mathscr{B}$ as follows. For a fragment F in arborescence $\mathscr{A}$, we define

$$\mathscr{L}(F) = [\mathrm{left}(F), \mathrm{left}(F) + |F| - 1],$$

where

$$\mathrm{left}(F) = \begin{cases} 0 & \text{if } F \text{ is the root of } \mathscr{A}, \\ \mathrm{left}(A) + \mathrm{indent}(A, F) & \text{if } (A, F) \text{ is in } \mathscr{A}. \end{cases} \tag{7}$$

For simplicity, this lays out every contig from position zero. It should be understood that two fragments overlap in the resulting layout only if their intervals intersect, and they are in the same arborescence. Layout $\mathscr{L}$ can be computed in linear time by evaluating (7) top-down from the roots of $\mathscr{B}$.

Recall from Section 2 that edges in our overlap graph are weighted by likelihood function $L(l, d)$. Each edge represents an alignment of maximum $L(l, d)$, where l is the number of matches in the alignment, and d is the number of errors. For a perfect alignment, $L(l, 0) = l$. Thus, when $\varepsilon = 0$, the weight of an edge (A, B) is the length of the longest prefix of B that can be overlapped with A. If we write $w(\mathscr{B})$ for the total weight of the edges in a branching $\mathscr{B}$, then, for the case of no error, the length of the layout $\mathscr{L}$ induced by $\mathscr{B}$ is

$$|\mathscr{L}| = \sum_{F \in \mathscr{F}} |F| - w(\mathscr{B}).$$

Since $\sum_F |F|$ is a constant for any given input, a branching of maximum weight gives a layout of minimum length. We take our problem to be the following.

DEFINITION. The *maximum-weight dovetail-chain branching problem* (BRANCHING) is, given a directed graph (V, E) with edge weight function w, with edges classified as either dovetails or containments, find a dovetail-chain branching $\mathscr{B} \subseteq E$ maximizing $w(\mathscr{B})$.

Kececioglu [18] generalizes the correspondence between dovetail-chain branchings and layouts to the case of error, and reduces the sequence reconstruction problem to the maximum-weight dovetail-chain branching problem. The reduction requires two assumptions on fragment error: that error is evenly distributed, and that approximate matching is transitive at the input error rate. In particular he shows that, given these assumptions, an algorithm for BRANCHING yields a reconstruction feasible at error rate $2\varepsilon/(1-\varepsilon)$ that is at most a factor $1/(1-\varepsilon)$ longer than the shortest reconstruction feasible at error rate ε.

This reduction is also one way of showing that BRANCHING is NP-complete: RECONSTRUCT is NP-complete for $\varepsilon = 0$, while the above result states that, for this case, BRANCHING solves RECONSTRUCT exactly. Given that a polynomial-time algorithm for BRANCHING is unlikely, how can we find a maximum-weight dovetail-chain branching in practice? Our strategy is to relax the dovetail-chain constraint. We can compute a maximum-weight branching, which may or may not be dovetail-chain, in polynomial time. Moreover, branchings can be generated in order of decreasing weight. Hence, we generate branchings in order of weight until finding one satisfying the dovetail-chain constraint. Our premise is that for fragments at a low error rate, from a sequence with few repeats, few branchings have to be generated.

It should be emphasized, however, that this approach requires exponential time in the worst case. Figure 6 gives a simple example where the maximum dovetail-chain branching has a rank that is exponential in the number of vertices (though see Section 4.3 for techniques to deal with such graphs). Since few users can wait for an exponential number of iterations, we place a limit on the number of branchings generated. For each branching generated, we invoke a procedure that greedily repairs any defects it may have. Of the branchings that are generated, the repaired branching of maximum weight is returned as a solution. As the limit on iterations is a constant, and greedy repair is efficient, a dovetail-chain branching, not necessarily of maximum weight, is delivered in polynomial time. A slight variant of greedy repair is known to be an approximation algorithm for BRANCHING.

Biologists often have additional conditions on a solution, besides length of reconstruction, that are difficult to capture formally. In such circumstances, it is

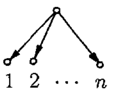

Fig. 6. A graph for which a maximum-weight dovetail-chain branching has exponential rank. All edges are dovetails of unit weight, so a dovetail-chain branching has only one edge, and is preceded by $\Omega(2^n)$ non-dovetail-chain branchings of greater weight.

desirable to see not one solution, but several, from which the truly best may be chosen. We show that alternate layouts are easily computed, and that our approach can accommodate various constraints. The next four sections present our algorithm for dovetail-chain branchings, and the fifth section describes our procedure for alternate layouts.

4.2. Generating Branchings. Efficient algorithms are known for maximum-weight branchings, and for generating branchings in order of decreasing weight. A maximum-weight branching over a graph of E edges and V vertices can be computed in $O(E + V \log V)$ time, as shown by Gabow *et al.* [11]. The K branchings of greatest weight can be generated in $O(KE \log V)$ time and $O(K + E + V)$ space, as shown by Camerini *et al.* [3]. Our method of generating branchings is similar to that of Camerini *et al.*, who apply the $O(E \log V)$ branchings algorithm of Tarjan [37], but has some differences. These differences are due to our particular application, namely, generating branchings to meet a dovetail-chain constraint, and allow us to apply the faster branchings algorithm of Gabow *et al.*, to generate K branchings in $O(K(E + V \log V))$ time.

4.2.1. Forming Constraints. Suppose we have computed a maximum-weight branching, which is not dovetail-chain. Such a branching contains a pair of dovetail edges e and f that leave a common vertex, or a containment edge e that is followed by a dovetail edge f. In both cases, we say *e conflicts with f.*

No dovetail-chain branching $\mathscr{B} \subseteq E$ contains both e and f. Either $\mathscr{B}$ contains neither e nor f, or f but not e, or e but not f. This can be expressed as two disjoint conditions:

1. $\mathscr{B} \subseteq E - \{e\}$ or
2. $\mathscr{B} \subseteq E - \{f\}$ and $\{e\} \subseteq \mathscr{B}$.

In the first case, we can continue by searching for a maximum-weight branching over the graph $(V, E - \{e\})$. In the second case, where $e = (A, B)$ is part of the solution, we can remove all out-edges from A, all in-edges to B, merge A and B into a single vertex to obtain graph $(\tilde{V}, \tilde{E})$, and continue by searching over $(\tilde{V}, \tilde{E} - \{f\})$, while recording e as part of the solution.

By solving both problems recursively, and returning the branching of greater weight, we will find a solution to the original problem. Since the subproblems are of smaller size, we have made progress. Since they partition the solution space, together they give a solution to the larger problem.

Further refining one of the subproblems gives three problems, then four problems, and so on. In general, at any point in branching generation, we have a collection of subproblems that partition the space of dovetail-chain branchings. Each subproblem is represented by two sets of edges: an *in-set*, which must be contained by the dovetail-chain branching, and an *out-set*, which must not be contained. Also associated with each subproblem is the weight of the heaviest branching satisfying the in- and out-set constraints. This weight is an upper bound on the solution value for the subproblem.

An iteration of the generator involves finding a subproblem $\mathscr{P}$ of greatest associated weight, and computing a maximum-weight branching $\mathscr{B}$ meeting $\mathscr{P}$'s constraints. If $\mathscr{B}$ is dovetail-chain, it is an optimal solution to the original problem, and we halt. ($\mathscr{B}$ has weight as great as any solution to a subproblem.) If $\mathscr{B}$ is not dovetail-chain, a pair of conflicting edges $\{e, f\} \subseteq \mathscr{B}$ is located, and $\mathscr{P}$ is split into two subproblems. Let I and O be the in- and out-sets for $\mathscr{P}$. One subproblem receives constraints I and $O \cup \{e\}$, and the other receives constraints $I \cup \{e\}$ and $O \cup \{f\}$. This follows a general method of Lawler [20] for generating next-best solutions to combinatorial optimization problems.

The resulting collection of problems is conveniently represented by a *computation tree*. Each node in the tree contains an in-list and an out-list, which consist of the edges in the in- and out-sets, along with the weight of the heaviest branching meeting these edge constraints. Internal nodes have two children, which refine their parent's subproblem. Leaves encode the current partition of the solution space. A heap of leaves prioritized by weight allows us to find a subproblem of greastest upper bound by extracting a leaf of maximum priority.

4.2.2. Computing Constrained Branchings. A branching of maximum weight meeting the in- and out-set constraints can be found by transforming the problem into one with no constraints. This transformation has two steps.

First, instead of solving a branchings problem, we solve a rooted spanning-arborescence problem. A *rooted spanning-arborescence* is a branching where every vertex, other than a specified root, has an in-edge. The maximum-weight branching problem can be reduced to the maximum-weight rooted spanning-arborescence problem by adding an artificial root to the graph, and adding edges of zero weight from the root to every vertex in the original graph. Choosing an edge from the root to vertex v in a rooted spanning-arborescence means no in-edge to v is chosen in the corresponding branching. The branchings algorithms of Camerini *et al.* [3] and Gabow *et al.* [11] in fact compute maximum-weight rooted spanning-arborescences.

The second step of the transformation removes the edge constraints. Every edge in the out-set is removed from the rooted graph, and for every edge (A, B) in the in-set, all edges of the form (C, B) are removed, where A and C are distinct. Clearly, the set of unconstrained arborescences over this new graph is the same as the set of constrained arborescences over the original graph.

4.2.3. Time and Space. The method we have outlined, which is essentially the method of Camerini *et al.*, simplified by the fact that we can more easily identify the edges e and f on which to decompose a subproblem, can be implemented efficiently in terms of the number of iterations, and the size of the original graph.

Each branching generated requires at most three constrained branching-computations, two heap insertions, and one heap deletion. One branching computation is for recovering the branching that meets the upper bound for the chosen subproblem, and the other two are for bounding the weight of its children when they are inserted into the heap. Each maximum-weight branching computation involves reducing the constrained branching problem to an uncontrained arborescence problem. The reduction takes $O(V)$ time, and computing a maximum-weight rooted spanning-arborescence takes $O(E + V \log V)$ time [11].

The time for heap operations can be bounded as follows. The heap contains leaves of the computation tree, and generating a branching creates one leaf. Thus the heap is of size $O(K) = O(2^E)$. Heap insertions and deletions take time logarithmic in the size of the heap, so the time per iteration for heap operations is $O(\log 2^E) = O(E)$.

Combining this with the above, the time per iteration is $O(E + V \log V)$, so the time to generate K branchings is $O(K(E + V \log V))$.

Space is required for the constrained branchings-algorithm, the computation tree, and the heap. Computing a branching takes $O(E + V)$ space [11]. The heap uses constant space per leaf, or $O(K)$ space in all. The computation tree appears to require $O(KE)$ space—it has $O(K)$ nodes and each node has an in- and out-list of size $O(E)$—but this can be reduced to constant space per node using the following idea of Gabow [10].

The in- and out-sets for a left child l in the computation tree may be obtained from its parent p, by adding one edge e to p's out-set, to form l's out-set, and by copying p's in-set. The in- and out-sets for a right child r may be obtained from its parent p and left brother l, by adding one edge f to p's out-set, to form r's out-set, and by adding l's edge e to p's in-set, to form r's in-set. This being the case, instead of storing two edge lists at a node, we can store pointers to its parent, its left brother, and the edge it adds to its parent's out-set. Following these pointers back to the root, we can recover the in- and out-sets for a node in $O(E)$ time using constant space per node. This gives a total of $O(K)$ space for the tree. With this representation, the space for generating K branchings is $O(K + E + V)$.

4.3. Accelerating Convergence. We now present two optimizations that accelerate convergence to a dovetail-chain branching. The first optimization addresses edge conflicts.

Before computing a branching for a problem with in-set I and out-set O, we add to O all edges in the graph that conflict with an edge of I. Certainly this is correct, as no dovetail-chain branching for the problem can contain any of these edges. The edges conflicting with a given set can be computed in $O(E + V)$ time. In the example of Figure 6, this reduces the number of branchings generated from $O(2^n)$ to $O(n)$.

The second optimization addresses an inherent redundancy in branching generation. We generate branchings as a means of generating layouts, but because the relation from branchings to layouts is many-to-one, several generated branchings can result in the same layout. Factoring out this redundancy requires a modification to the computation-tree data structure, and a more careful method of identifying edge conflicts.

We capture the set of edges in all branchings inducing the same layout as a given branching $\mathcal{B}$, by the *closure* of $\mathcal{B}$. Informally, this set contains all edges in the graph that overlap fragments in the same relative position as $\mathcal{B}$. Formally, the closure of a branching $\mathscr{B}$ in overlap graph G is the set

$$\text{closure}(\mathscr{B}) = \{(A, B) \in G \text{ such that } |\text{indent}(A, B) - (\text{left}_{\mathscr{B}}(B) - \text{left}_{\mathscr{B}}(A))| \\ \leq \varepsilon|A| + \varepsilon|B|\}.$$

(In this definition, it is understood that (A, B) is in the closure only when A and B are in the same arborescence of $\mathscr{B}$.) In words, we measure the difference between the placement of A and B in the layout induced by $\mathscr{B}$, and in the overlap (A, B). If this difference can be explained by the error rate, (A, B) is in the closure. Given a branching, we can compute its closure in $O(E + V)$ time: determining the layout takes $O(V)$ time, after which testing an edge for membership takes $O(1)$ time.

Removing (A, B) from a branching, and replacing it with (C, B) from the closure, gives in essence the same layout. This is exactly what we want to avoid. Formally, let the *kin* of an edge (A, B), with respect to branching $\mathscr{B}$ and graph G, be the set

$$\text{kin}_{\mathscr{B}}(A, B) = \{(C, B) \in G \mid C \neq A\} \cap \text{closure}(\mathscr{B} \cup \{(A, B)\}).$$

We form $\text{kin}_{\mathscr{B}}(A, B)$ only when $\mathscr{B} \cup \{(A, B)\}$ is a branching, so that the closure is well defined. The kin of (A, B) can be computed in $O(V)$ time: determining the layout for the closure takes $O(V)$ time, after which we only need to examine the $O(V)$ edges that enter B, and agree with the layout.

We use kin as follows. Before computing a branching for a problem $\mathscr{P}$ with in-set I and out-set O, that adds (A, B) to the out-set of its parent, we augment O with $\text{kin}_I(A, B)$. We known (A, B) was removed from the branching of $\mathscr{P}$'s parent, which contained set I, because it created a conflict. Adding $\text{kin}_I(A, B)$ to O prevents the selection of another edge that places B in the same position. Note that obtaining O involves recovering the out-set of P's parent, which requires computing another kin-set, which involves another out-set, and so on up the tree. Computing all kin-sets could take $O(KV)$ time.

To retain the $O(E)$ time complexity for recovering in- and out-sets, we modify the computation tree. A node now stores, along with the edge e that it adds to its parent's out-set, a kin-list for e. This list contains $\text{kin}_I(e) - O$, where I is the in-set for the node and O is the out-set for its parent.

To recover an out-set for a node, we follow node pointers back to the root, as before, but now we also copy the kin-lists of the nodes visited. We recover in-sets as before. Recovering in- and out-sets then takes time $O(E)$. When creating a node, we recover its in-set I, its parent's out-set O, compute $\text{kin}_I(e)$, and store $\text{kin}_I(e) - O$ at the node, all in time $O(E + V)$. Thus the second optimization does not increase the time complexity of branching generation.

Unfortunately, the space complexity increases to $O(KV)$ in the worst case, as the tree has $O(K)$ nodes and each kin-list can have $O(V)$ edges. In practice, however, kin-sets have constant size, as the in-degree of vertices is bounded by a constant (see Section 6.4). We call such graphs *sparse*. For sparse graphs, the space for all kin-lists is $O(K)$, and the space complexity of branching generation does not increase.

4.4. Resolving Conflicts. When incorporating these optimizations, we must be careful, since a representative branching is being chosen for a layout. Consider two branchings, $\mathscr{B}_1 = \{A \to B, A \to C\}$ and $\mathscr{B}_2 = \{A \to B, B \to C\}$, that induce the same layout. Only $\mathscr{B}_2$ is dovetail-chain. If the representative that is generated for the layout is $\mathscr{B}_1$, a consistent layout will be rejected.

Consequently, it no longer suffices to test whether or not a generated branching $\mathscr{B}$ is dovetail-chain. We must test whether there is a dovetail-chain branching $\tilde{\mathscr{B}}$ over the graph that induces the same layout as $\mathscr{B}$. If such a $\tilde{\mathscr{B}}$ exists, we say it *resolves* the conflicts in $\mathscr{B}$, and we return $\tilde{\mathscr{B}}$ and halt.[11] If no $\tilde{\mathscr{B}}$ exists, the conflict that we use to generate subproblems must be one that cannot be resolved. We now present a procedure for identifying irresolvable edge conflicts. As a side effect, it will find a resolved branching $\tilde{\mathscr{B}}$ when one exists.

Given a branching $\mathscr{B}$ with conflicts, we compute its induced layout $\mathscr{L}$, and closure $\mathscr{C}$. Over the containment edges in $\mathscr{C}$, we compute a maximum-weight branching $\tilde{\mathscr{B}}_1$. Every fragment that is contained by an edge of $\tilde{\mathscr{B}}_1$ is removed from consideration. The remaining fragments are roots of $\tilde{\mathscr{B}}_1$, and we sort them within contigs by increasing left endpoint in $\mathscr{L}$. For consecutive pairs of fragments A, B, we look for a dovetail $A \to B$ in $\mathscr{C}$. Call the set of dovetails that are found, $\tilde{\mathscr{B}}_2$. If no dovetail exists for some consecutive pair of fragments, $\mathscr{B}$ contains an irresolvable conflict. Otherwise, resolved branching $\tilde{\mathscr{B}} = \tilde{\mathscr{B}}_1 \cup \tilde{\mathscr{B}}_2$ is returned. Note that $\tilde{\mathscr{B}}$ is dovetail-chain.

This procedure is correct, as it can be shown that a contained fragment in $\mathscr{L}$ must have a containment in-edge in $\tilde{\mathscr{B}}$, and the remaining fragments must be related by a chain of dovetails in $\mathscr{C}$. This chain is unique, as two distinct chains through the same set of fragments would create a dovetail cycle, which is impossible for a given layout. Thus, if there is a dovetail-chain branching in the graph that yields the given layout, our procedure finds it. Moreover, the resolved branching is unique up to containment edges. By using containments of maximum weight, the procedure finds an optimal resolved branching.

If a dovetail chain is not found, what pair of edges in $\mathscr{B}$ forms an irresolvable conflict? Certainly the first consecutive pair A, B with no dovetail $A \to B$ in $\mathscr{C}$ is the source of a conflict in $\mathscr{B}$. Depending whether or not one fragment is a descendant of the other, there are two cases.

Suppose, without loss of generality, that B is a descendant of A, and let P be the path from A to B in $\mathscr{B}$. Since B is not contained in $\mathscr{C}$, path P must end in a dovetail edge e. Since A and B are adjacent in the layout, and $(A, B) \notin \mathscr{C}$, path P must begin with a containment f. We choose for our conflicting pair $\{e, f\}$. It is irresolvable, as there is no edge $(A, B) \in \mathscr{C}$.

Now suppose neither A nor B is a descendant of the other. Let P be the path from their common ancestor to A, and let Q be the path from this ancestor to B. Since A and B are not contained in $\mathscr{C}$, both P and Q must end in a dovetail edge.

[11] In general, we cannot claim that $w(\tilde{\mathscr{B}}) = w(\mathscr{B})$, hence $\tilde{\mathscr{B}}$ may not be a maximum-weight dovetail-chain branching. In the absence of error, however, the overlaps in $\tilde{\mathscr{B}}$ must be as long as the overlaps in $\mathscr{B}$, which does imply $w(\tilde{\mathscr{B}}) = w(\mathscr{B})$. Thus, for low error rates, it is reasonable to assume that the weight of $\tilde{\mathscr{B}}$ is close to the weight of $\mathscr{B}$. In such a situation, $\tilde{\mathscr{B}}$ cannot be far from optimal, and is certainly worth reporting. Moreover, in Section 4.6 we discuss how to generate alternates.

Let these final edges be e and f. Since $(A, B) \notin \mathscr{C}$, this pair is again an irresolvable conflict.

In both cases, the conflicting pair can be located in $O(V)$ time by walking up $\mathscr{B}$. Determining layout $\mathscr{L}$ takes $O(V)$ time, and forming closure $\mathscr{C}$ takes time $O(E)$. Sorting the fragment intervals by left endpoint takes $O(V \log V)$ time. Verifying the dovetail-chain takes time $O(E)$. Thus, we can find a resolved branching, or an irresolvable conflict, in $O(E + V \log V)$ time, which is within the complexity of branching generation.

Before moving on to the next section, we review how the closure optimizations and conflict resolution are incorporated into our generator.

An iteration consists of removing a problem of greatest upper bound from the heap, recovering the branching meeting the bound, and splitting the problem into two subproblems. To recover a branching, we determine its in- and out-sets, and compute a maximum-weight branching that meets these constraints. To determine the constraints we walk up the computation tree, collecting an in-set I and an out-set O. We augment O with the kin-sets encountered during the walk, and all edges conflicting with I. The recovered branching is tested for irresolvable edge conflicts. If none exist, an equivalent dovetail-chain branching is returned, and we halt. Otherwise, an irresolvable conflict is identified, two subproblems are placed in the heap, and we iterate. A dovetail-chain branching is delivered in $O(K(E + V \log V))$ time, where K is the number of iterations. Space is $O(KV + E)$ worst case, and $O(K + E + V)$ for sparse graphs.

4.5. Repairing Irresolvable Conflicts. On every iteration that fails to produce a dovetail-chain branching, conflicts in the generated branching $\mathscr{B}$ are repaired by a greedy procedure to give a dovetail-chain branching $\hat{\mathscr{B}}$. Of these repaired branchings, one of maximum weight over all iterations is retained. In the event that the generator exceeds the limit on iterations, we return a maximum-weight repaired branching.

To repair a non-dovetail-chain branching $\mathscr{B}$, we locate its forbidden subgraphs and remove their edges. Note that the resulting branching $\hat{\mathscr{B}}$ is dovetail-chain. Edges in $E - \hat{\mathscr{B}}$ are ordered by decreasing weight, and considered for inclusion in $\hat{\mathscr{B}}$. If including an edge in $\hat{\mathscr{B}}$ preserves the branching property, and the dovetail-chain property, it is added to $\hat{\mathscr{B}}$. After all edges have been considered, the final $\hat{\mathscr{B}}$ is returned.

Locating and removing the forbidden subgraphs of $\mathscr{B}$ takes $O(V)$ time. Sorting the edges in $E - \hat{\mathscr{B}}$ takes $O(E \log V)$ time. Testing an edge for the dovetail-chain property can be done in $O(1)$ time by maintaining two boolean variables for each fragment. One variable records whether the fragment has a containment in-edge in $\hat{\mathscr{B}}$, and the other records whether it has a dovetail out-edge. Including containment $A \Rightarrow B$ preserves the dovetail-chain property if B has no dovetail out-edge, while including dovetail $A \rightarrow B$ preserves the dovetail-chain property if A has no dovetail out-edge, and no containment in-edge. Including an edge (A, B) preserves the branching property if B has no in-edge, and (A, B) does not create a cycle. Since (A, B) forms a cycle if and only if A and B are members of the same arborescence, we can test for cycle creation in essentially constant time by maintaining the partition of fragments into arborescences with disjoint sets [37]. Thus the dominant step is sorting the edges. In short, greedy repair can be performed in $O(E \log V)$ time worst-case.

Interestingly, it is asymptotically more expensive to repair a branching greedily than to compute one of maximum weight. This is in the worst case, however. For the sparse graphs of practice, $E = O(V)$, and the time for greedy repair is $O(V \log V)$.

We note that greedy repair is essentially a greedy algorithm for dovetail-chain branchings, started from a partial branching. In a sense, it is *partially greedy*, since the initial branching is obtained by a global optimization. Tarhio and Ukkonen [36] and Turner [38] analyze the totally greedy algorithm on overlap graphs with $\varepsilon = 0$, and show that it finds a solution of weight at least one-half the maximum. For overlap graphs with $\varepsilon \neq 0$, the tightest analysis we know for the totally greedy algorithm gives a factor of one-third. In fact, there are graphs for which totally greedy performs better than partially greedy, and vice versa [18].

Even so, we conjecture that the partially greedy algorithm achieves at least a factor of one-third. Moreover, the weight of the last branching generated is an upper bound on the weight of an optimal dovetail-chain branching. If we terminate without finding an optimal solution, we can report how far from optimal our solution is. This is not possible with a purely greedy strategy.

4.6. Producing Alternates. Sometimes the biologist has additional criteria for a reconstruction that are difficult to formalize or incorporate into an algorithm. The biologist may have a rough idea of the length of the solution, or know that a section of the reconstruction containing repeats is not correct. In short, the biologist may demand an alternate solution, and may wish to specify additional constraints.

4.6.1. Strongly Independent Alternates. We use the closure of a branching to generate alternate solutions. If closure $\mathscr{C}_1$ and $\mathscr{C}_2$ of two branchings differ, their layouts differ. Requiring in addition that $\mathscr{C}_1 \nsubseteq \mathscr{C}_2$, and $\mathscr{C}_2 \nsubseteq \mathscr{C}_1$, ensures that one layout is not simply part of the other. In general, suppose the closures of the first n solutions are $\mathscr{C}_1, \mathscr{C}_2, \ldots, \mathscr{C}_n$. We say the nth branching is *strongly independent* of the first $n - 1$ branchings, if

$$\mathscr{C}_n \nsubseteq \bigcup_{i<n} \mathscr{C}_i.$$

This guarantees that every branching induces a configuration not seen before. While we do not know how to generate strongly independent branchings on-line in order of weight, there are at most E of them, so we can afford to generate them all, and sort them.

To generate an alternate layout, we find the heaviest edge in the graph not in the union of the closures of the previous dovetail-chain branchings. We then invoke our generator with the constraint that the chosen edge e is contained in the branching. This simply involves adding e to the in-set at the root of the computation tree, and forcing greedy repair to retain e.

Generating M alternates takes $O(E \log V + M(E + V))$ time on top of branching generation. Before computing any alternates, we form a sorted list of all the edges in the graph. As each alternate is produced, we compute its closure, and remove its closure edges from the list. To produce the next alternate, we seed the branching generator with the edge at the head of the list. Sorting the edges takes $O(E \log V)$ time, and computing the closure and updating the list takes $O(E + V)$ time for each alternate.

Producing all M alternates in order of weight requires an additional $O(M \log M) = O(E \log V)$ sort, and can take $O(MV)$ additional space to store the branchings. The space can be reduced to $O(M + V) = O(E + V)$, at the cost of doubling the time, by storing only the M seed edges, and regenerating an alternate from its seed once their order has been established.

To summarize, M strongly independent alternates can be generated off-line, in order of weight, in $O(MK(E + V \log V) + E \log V)$ time, and $O(K + E + V)$ space, where K is the maximum number of branchings examined for an alternate.

4.6.2. User-Constrained Alternates. We can also produce alternates from constraints provided by the user. Biologists sometimes know the order in which a subset of the fragments should overlap, say from a directed sequencing method, or a restriction map. Othertimes, they may simply know that a configuration of fragments is incorrect, and wish to prevent it from appearing again. We can express some of this information with in- and out-sets of edges.

When the order of some fragements is known, we retain in the graph only those edges that are consistent with the ordering. All inconsistent edges are placed into the out-set. We note that this can fail to enforce a partial order on fragments. For example, if fragment B should follow fragment A, but nothing is known about fragment C, we cannot rule out $C \rightarrow A$ and $B \rightarrow C$ individually, yet together they form a path placing B before A. When a total order on some fragments is known, and no fragments are allowed in-between, we can enforce the order by placing a dovetail-chain into our in-set.

On the other hand, while the order for the layout may not be known, the biologist may know that what is given is incorrect. Here the user might select a portion of a contig in a generated layout, and ask that the fragments selected be completely rearranged. In this case, we would compute a closure from the sublayout on the fragments, and place these edges into our out-set. Note that this constraint is very severe, however.

Finally, if the user wishes to freeze a sublayout we can use our conflict resolution procedure of Section 4.4 to determine a branching inducing just the sublayout, and place these edges into our in-set.

5. Multiple Sequence Alignment.

At this stage we have a branching that specifies a consistent layout of the fragments. The output of our algorithm is a reconstructed sequence. In this final phase, we obtain a sequence from the branching by forming the closure of the branching, which consists of all overlaps that agree with the layout, merging these overlaps into a multiple sequence alignment, and voting on a consensus sequence for the alignment. This procedure can recover a sequence whose error is far less than that of any one fragment.

5.1. The Maximum-Weight Trace Problem. In the last section we defined the closure of a branching, which contains all edges in the graph that overlap fragments in the same relative position as the induced layout. The multiple sequence alignment[12] that we seek for a layout should agree with these overlaps. Exact agreement, however, is not always possible. We settle for a multiple alignment that is close to the pairwise alignments, and formalize a notion of closeness as follows.

An edge in the closure, that aligns $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$, can be represented by a list of pairs of positions, $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)$, where $1 \le i_1 < i_2 < \cdots < i_k \le m$, and $1 \le j_1 < j_2 < \cdots < j_k \le n$. A pair (i, j) matches characters a_i and b_j.

We treat each pair as a constraint on our multiple alignment, namely, that both characters must appear in the same column of the alignment. As Figure 7 shows, it may not be possible to satisfy all the constraints in a collection of pairwise alignments. We may have to settle for a subset of the constraints. To discriminate among subsets, we weight each constraint by the similarity of the pair of characters that are matched, and seek a subset that is satisfiable, and of maximum total weight.

To formalize when constraints are satisfiable, we define an *alignment graph* $(V, E, \prec)$, whose vertices V correspond to sequence characters, and whose edges E correspond to pairs of characters matched by the alignments. Over the vertices we define a partial order $\prec$. In this order, $v \prec w$ if v and w are both characters of a sequence S, and character v precedes w in S. Essentially, the order of characters between columns in any legal alignment must respect $\prec$.

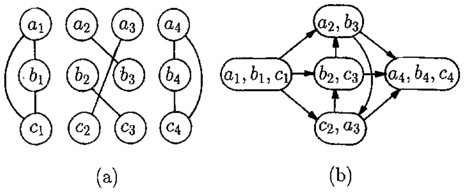

Fig. 7. Pairwise alignments may not form a multiple alignment. (a) Three pairwise alignments of $a_1a_2a_3a_4$, $b_1b_2b_3b_4$, and $c_1c_2c_3c_4$. Edges join matched characters. (b) The induced connected components, which contain a cycle under $\prec^*$.

[12] A *multiple sequence alignment* of sequences $S_1, S_2, \ldots, S_k$ is a matrix $A = (a_{ij})^{1 \le i < k, 1 \le j \le n}$ where row $a_{i1} \cdot a_{i2} \cdots a_{in}$ gives S_i. An entry a_{ij} may equal the *null character* ε, which is the identity under concatenation.

In an alignment graph $(V, E, \prec)$, every subset $T \subseteq E$ induces a collection of connected components[13] that partition V. For components X and Y, let $X \prec^* Y$ if there is an $x \in X$ and a $y \in Y$ such that $x \prec y$. Relation $\prec^*$ may not be a partial order, for it is possible to have both $X \prec^* Y$ and $Y \prec^* X$ when $X \neq Y$. When relation $\prec^*$ on the components of T is a partial order, the constraints of T are satisfiable: every component corresponds to a column of the alignment, and any topological ordering[14] of the components that respects $\prec^*$ is a valid order for the columns.

A topological order exists precisely when $\prec^*$ does not contain a cycle. In other words, the constraints of T are satisfiable if and only if $\prec^*$ on the components of T is acyclic. We call a satisfiable set T, a *multiple sequence trace*. This generalizes the standard notion of trace in sequence comparison [31, p. 12].

Given a trace, we can form a multiple sequence alignment by determining its connected components, and topologically sorting them. For a trace of M edges over a graph of N vertices, finding the connected components takes $O(M + N)$ time. The topological sort takes time linear in the size of the relation, which can be represented by less than N ordered pairs. Thus we can recover a multiple alignment from a trace in $O(M + N)$ time. Since simply reading the input and outputting the alignment requires $\Omega(M + N)$ time, we concentrate on finding traces, rather than computing alignments. Our problem is the following.

DEFINITION. The *maximum-weight trace problem* (TRACE) is, given an alignment graph $(V, E, \prec)$, with edge weight function w, find a trace $T \subseteq E$ maximizing $\sum_{e \in T} w(e)$.

TRACE is NP-complete [18], and remains so even when, as in our application, the edges between any two sequences form an alignment, and the length of every sequence is bounded by a constant.

We next present a fast heuristic for maximum-weight trace, and in Section 5.3, adapt it to the instances that arise in sequence reconstruction.

5.2. A Fast Heuristic. Given the NP-completeness of TRACE, our strategy is to design an algorithm that is fast, and in practice delivers near-optimal traces. Our heuristic is based on the well-known observation that a tree of pairwise alignments is a trace.[15] In other words, given an alignment graph G, consider choosing a pairwise trace between every two sequences. A tree over the sequences, consisting of pairwise traces, gives a multiple sequence trace for G.

Note that, in our alignment graphs, the edges between any two sequences

[13] The *connected components* of (V, E) induced by $F \subseteq E$ are the maximal sets $C \subseteq V$ such that every pair of vertices in C is connected by a path in F. Maximal means there is no vertex outside C connected to a vertex in C by F.

[14] A *topological order* for a set S with partial order $\prec$ is a total ordering of the elements of S that respects $\prec$.

[15] This observation, expressed differently, can be found in many papers. Perhaps the first occurrence is in [29].

already form a pairwise trace. Since any tree of these pairwise traces gives a multiple sequence trace, a simple heuristic is to choose a tree of maximum total weight.

We can apply this as follows. The pairwise traces from which we select a tree correspond to the overlaps in the closure of our branching. Treat each overlap as an undirected edge, and weight it by the sum of the similarities of the characters matched in the overlap. (Section 3.3.5 defines the similarity measure we use.) Over these edges, compute a maximum-weight spanning tree.

Computing the closure takes $O(E + V)$ time for an overlap graph of E overlaps and V fragments. For overlaps with a total of M pairs of matched characters, weighting the overlaps takes $O(M)$ time. For a closure of $\tilde{E}$ overlaps, computing a maximum-weight spanning tree takes $O(\tilde{E} + V \log V)$ time [9]. This delivers a trace in $O(E + M + V \log V)$ time.

Kececioglu [18] shows the resulting trace has weight at least $2/V$ of the maximum. This bound is tight, but pessimistic. Often the alignment graphs that arise can be partitioned into subgraphs of at most D fragments, where $D \ll V$ is the maximum number of fragments that mutually overlap in the layout. We call D the *coverage depth* of the layout, which in practice is a constant, usually between 5 and 10. On such inputs, the multitrace is within a factor of $2/D$ of optimal. For a coverage depth of 6, this means the heuristic achieves a factor of one-third.

Instances that meet even the coverage-depth bound appear unlikely to occur in practice. Real data has few errors, which lends structure to the pairwise traces. We can refine the heuristic to take advantage of this structure.

5.3. A Sliding-Window Variation. At the low error rates of current practice, our alignment graphs have a regular underlying structure. When no error is present, the sequences are identical, and the alignment graph is a series of columns, each column a complete subgraph. When a rare error is present, its effect on this structure is to displace or delete some edges local to the defect. For such graphs, most edges in a pairwise trace (A, C) will coincide with the trace of A and C induced by pairwise traces (A, B) and (B, C). In other words, the structure of pairwise traces tends to be transitive, due to high edge transitivity in the near-complete subgraphs. In such a situation the heuristic performs well, since most trees of pairwise traces induce an alignment of near-optimal weight.

We can improve these alignments further by adapting to local variation in sequence similarity. Errors tends to cluster at fragment ends, so instead of using one tree across a fragment, we allow the tree to adapt as errors arise, switching to a tree that favors similar sequences. To do this, we start with the alignment produced by the heuristic. A window containing a fixed number of columns is then swept across the alignment. The width of the window is a parameter to the algorithm. Over the alignment subgraph defined by the characters in the window, we compute a maximum-weight spanning tree. The first column of the alignment induced by the tree is output. These characters are removed from the window, and the column to the right of the window is added. This advances the window, and the process is repeated.

The window itself is represented by a *window graph.* This graph consists of supervertices, which represent the sequences spanned by the window, i.e., sequences with a character in the window, and superedges, which represent pairwise traces between the spanned sequences. The restriction of the alignment graph to the window is given by a left and a right boundary. These boundaries are the position of the leftmost character in the window, and the rightmost character in the window, for each spanned sequence. Supervertices are attributed with a left position, for the character just inside the left boundary, and a right position, for the character just outside the right boundary.

We assume for now that each pairwise trace is given by a list of matches in left-to-right order. Superedges are attributed with a left match, which points to the first match on the list inside the left boundary, and a right match, which points to the first match outside the right boundary. Superedges also store the total weight of the matches between the left and right boundaries.

In this sliding-window variation, the window can be viewed as a generator of columns. The next three sections describe the steps in column generation: how we update the window when advancing the left boundary, how we advance the right boundary, and how we compute a spanning tree over the window graph incrementally. Many details are bookkeeping, but are given for completeness. The final sections specify our representation of pairwise traces, our similarity function for weighting trace edges, our voting function for consensus characters, and a column-compression optimization.

5.3.1. Advancing the Left Boundary. A spanning tree T over the window graph selects pairwise traces whose matches form a multiple sequence trace on the characters in the window. These matches form connected components of characters in the alignment graph. The column generated by the window is an initial component under $\prec^*$ of characters on the left boundary. Given our window representation, we can find an initial component by a depth-first search over T.

We first order the children in T left-to-right according to the layout, and pick for the root of T the leftmost fragment in the layout. Our depth-first search traverses this tree, passing up the initial component in an alignment over the subtrees. The component is represented by a list of pointers to sequences; the nonnull characters in the column are the left boundary characters of the sequences on this list. Spanned sequences not on the list contribute null characters to the column.

When the search visits a node A of T, the component list for the subtree is initialized to sequence A. Its children are then examined in order. For child B, the left match in the superedge to A is examined. We identify four cases, as illustrated in Figure 8. Cases (c) and (d) are handled together.

Case (*a*). (The match touches the left boundary character of A, but not B.) In this case, the boundary characters of A and B are in different components. Furthermore, the component containing B must precede the component containing A, since the boundary character of B precedes the character matched with A.

In this situation, we recursively search the subtree rooted at B. When this search returns and passes up a component $\mathscr{C}$, we short-circuit the search of A's children and simply return $\mathscr{C}$.

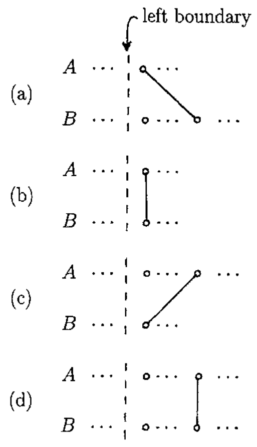

Fig. 8. Left matches of A and B. Cases (a)–(d) are distinguished by whether or not the match touches a left boundary character of the window.

Case (*b*). (The match touches the boundary characters of A and B.) In this case, A and B are in the same component. Recursively search the subtree rooted at B, and let $\mathscr{C}$ be the component returned by the search. If B is a member of $\mathscr{C}$, we append the members of $\mathscr{C}$ onto A's list, and continue the search from the next child of A. Otherwise, $\mathscr{C}$ precedes the component containing B. In that case, we can short-circuit the search from A, and simply pass up $\mathscr{C}$.

Cases (*c*) *and* (*d*). (The match touches either the boundary character of B and not A, or neither B nor A.) Again the components of A and B are distinct, but now either A precedes B, or they are incomparable.

In either case, we retain A's component for the generated column, and instead of searching B's subtree, continue the search from A's remaining children.

The component returned from the root gives the column that is generated.

We can account for the time to find the initial component by charging operations to the edges of T. The operations are concatenating component lists, and testing whether a child is in the component it passes up. With doubly linked lists, concatenation can be performed in constant time. Each node knows whether it is in the component it passes up; by passing this information onto its parent, a test can be performed in constant time.

Charging these operations to the edge from a child to its parent, each edge is charged a constant amount of time. The time then to compute a column is linear in the size of T, which, for a layout of coverage depth D, is $O(D)$.

Having determined the first column of the alignment in the window, we advance the left boundary. We increment the left position of each sequence with a character in the column, which effectively removes the character from the window. If this character is the last in the sequence, we delete its supervertex and all incident superedges from the window graph. Otherwise, we retain the supervertex and update its incident superedges. If the left match of a superedge involves the character in the column, we advance the left match pointer to the next match in the pairwise trace, and decrease the weight of the superedge by the match weight. All this can be done in time linear in the size of the window graph. Section 5.3.3 describes how we determine the tree T.

5.3.2. Advancing the Right Boundary. Advancing the right boundary involves adding a column on the right. We determine this column exactly as we determined the first column in the window, namely by traversing a tree over the sequences depth-first, except now the tree is fixed by the branching.

Once we have determined the column, we examine its characters. If the character is from a new sequence (one that is not in the window) we create a supervertex for the sequence. All overlaps in the closure incident to the sequence are examined, and if the alignment is with a sequence in the window graph, a superedge is created. Since each sequence is inserted into the window graph once, the total time for insertion is linear in the total number of fragments and overlaps in the closure. When a superedge is created, its weight is initialized to zero, and the left and right match pointers are set to the first match of the pairwise trace.

After creating any new supervertices and edges, the right position is incremented for every sequence with a character in the new column. We also examine the superedges incident to these sequences. If the right match in the superedge touches characters that are both in the window, we increase the weight of the superedge by the weight of the match. If either character in the right match is in the window, we advance the right match pointer to the next match in the trace.

When we remove a column from the left and add one on the right, we try to maintain the window at roughly the same width. If we only ever add one column, the window can shrink. This will happen, for example, when the column removed contains a character from every sequence, and the column added contains only one character. On the other hand, if we always add a column, the window may expand. This happens when the column removed contains few characters, and the column added contains many. To maintain the size of the window, we use the following rule. A column is added while half the spanned sequences that extend beyond the right boundary have less than w characters in the window, where w is the window width. Sometimes this rule adds no columns, and sometimes it adds many. It ensures that the majority of the sequences are at the window width, and tends to maintain the volume of the window. Checking the rule takes $O(D)$ time for D spanned sequences. Since determining the column on the right takes at least this much time, it does not increase the time complexity.

5.3.3. Computing the Spanning Tree. The tree we use to determine the first column in the window is a maximum-weight spanning tree over the window graph. As we noted in Section 5.2, a maximum-weight spanning tree can be found in $O(E + V \log V)$ time for a graph of V vertices and E edges. In practice, however, we recommend a different spanning-tree algorithm. The $O(E + V \log V)$-time algorithm requires a Fibonacci heap [9], or in practice a pairing heap

[8]. Since a spanning tree is computed for every column, the overhead of these data structures is unappealing.

Moreover, our spanning-tree problems often vary only slightly from window to window, in which case it is unnecessary to compute a tree from scratch. When a column is emitted that contains few characters, as is the case when an insertion error has occurred, the weight of only a few edges is changed. Incremental spanning-tree algorithms are available that can quickly recompute an optimal tree after the weight of one edge in the graph has changed. However, when a column is emitted that contains many characters, all identical, as is the case when a deletion error has occurred, the weight of nearly every edge in the graph is changed, all by the same amount. This can require $O(E)$ invocations of an incremental algorithm to arrive at the same tree as the initial one. An incremental algorithm that avoids the overhead of a pairing heap, while performing gracefully at both extremes, is preferable.

A well-known algorithm with these properties is Kruskal's. Recall that this algorithm starts from the empty tree, sorts the edges by nonincreasing weight, and, considering them in order, adds an edge to the tree if it does not create a cycle. This can all be accomplished in time $O(E \log V)$, the complexity being dominated by the edge sort. Notice, however, that if we retain the sorted edge list from the previous column, this list will be partially sorted for the current column. An insertion sort, for example, on an n-element list, with k elements or m pairs of elements out of order, takes $O(n + m) = O(kn)$ time.[16] In the common case where few edge weights have changed, or nearly all weights have changed uniformly, this is fast.

5.3.4. Representing Pairwise Traces. Other than the lists of matches for the pairwise traces, all data structures are linear in the size of the window graph. (In fact, the sliding window algorithm never constructs the entire alignment graph or multiple alignment matrix.) Except for matches, this is linear in the number of fragments and overlaps in the closure.

The space for matches can be kept small by representing pairwise traces with edit scripts. A script specifies the insertions, deletions, and substitutions to edit one sequence into the other. For low error rates and long sequencies, this is a substantial saving over a list of matched characters.

Edit scripts change the algorithm slightly, as our basic operation on an alignment is to ask for the match at a given position. Notice, however, that these queries come left-to-right across a sequence. (For a window boundary, we only need to deliver the first match to the right of the boundary, and update this match when the boundary is advanced.) We can represent an edit scrpt with a pair of vectors giving the ascending positions of unmatched characters in both sequences. For each superedge, we maintain the pair of positions at the current

[16] The time can be reduced to $O(n + k \log(m/k)) = O(n + k \log n)$ using a balanced tree to perform the insertions [24, pp. 222–224], though this is unnecessary for our small window graphs.

match, and two pointers into the edit script giving the next unmatched character in both sequences. With this representation, the next match on the boundary can be delivered and updated with no increase in time or space complexity. Finding the next match may require skipping over several unmatched characters, but the total time is proportional to the number of alignment columns, and the length of the sequences.

5.3.5. Weighting Trace Edges. Biologists often denote the nucleotides in the DNA sequence for a fragment with *ambiguous base codes*. An ambiguous base code is a subset of {a, c, g, t}, and represents the set of possible nucleotides at a sequence position that cannot be resolved uniquely.

Each character in our sequence then is really a set of letters. We can encode each set with a bit-vector, since only four bits are needed for the DNA alphabet, and each vector will fit in a byte. When computing pairwise alignments during overlap graph construction, we consider two characters to match if their encoded sets intersect. Intersection can be testd in constant time with a bitwise-and operation.

When computing multiple sequence alignments, we favor more precise matches by giving such trace edges more weight. The weight of an edge between characters with sets X and Y is

$$|X \cap Y|/|X \cup Y|.$$

The denominator is never zero, since X and Y are never empty. This gives an exact match unit weight, an ambiguous match somewhat less, and a complete mismatch zero weight.

We can compute the weight of a trace edge in $O(1)$ time. The intersection and union can be computed with bitwise-and and bitwise-or operations, and the cardinality of the resulting set can be computed by table look-up.

5.3.6. Voting on a Consensus. Our reconstructed sequence is determined by consensus from the multiple sequence alignment. Each character in a column places a vote for each letter in its set, where a null character is equivalent to the empty set. For each letter in the alphabet, the votes are tallied. The consensus character is the set of letters that receive at least $n/2$ votes, where n is the number of sequences spanned by the column.

This rule minimizes the total number of insertions, deletions, and substitutions to convert the consensus character into its column. For a layout of coverage depth D, the time to determine such a character is $O(D)$.

5.3.7. Compressing Columns. As described in Section 5.3.1, two characters are placed in the same column only when there is a path of matches that join them in the trace given by the spanning tree. This policy can overlook matches outside the tree that join characters in adjacent columns. Since we compute a consensus from the multiple alignment by voting, we want to merge adjacent columns whenever possible. If characters are spread across several columns, their vote gets divided. In the extreme, their vote may be sufficiently divided to prevent them from appearing in the consensus, which can cause a deletion error in the reconstruction.

We can prevent this to some extent as follows. Generated columns are filtered through a *press* before being output. This data structure contains a set of sequences from the window graph. For each sequence in the set, the press holds a nonnull character; together these characters form a column not yet output.

Recall that we also represent a column generated from the window as a set of sequences with nonnull characters. When a new column is generated, we compare it with the one in the press. If the sets are disjoint, we form their union, effectively compressing the two columns. Otherwise, there is some sequence with a nonnull character in both the press and the generated column. In this case, we output the column in the press, and replace it with the one from the window.

Column compression takes time proportional to the number of nonnull characters in the columns that enter and leave the press, which does not increase the time complexity for multiple alignment.

To summarize, given an alignment graph of N vertices, induced by a closure of E overlaps from a layout with coverage depth D, the sliding-window algorithm computes a multiple sequence alignment of L columns in $O(N + D^2L)$ time, and $O(D + E + \varepsilon N^2)$ space.

6. Experimental Results.

To explore the viability of this approach to sequence reconstruction, we have implemented a software package embodying the preceding suite of algorithms [19]. In both the orientation and layout phases, exact algorithms are run first. If the size of a search tree becomes too large, for example when $K = 1000$, the phase switches to the approximation algorithm to produce a solution. This section presents results from tests of this implementation on simulated sequencing data. By using simulated data, we could be certain of the correct solution to each test. We also emphasize that these experiments do not constitute an exhaustive or conclusive study. Our goal was simply to get a feel for the performance of our methods.

The experiments were conducted as follows. Given a sequence, we sampled it at randomly chosen intervals to form a collection of substrings. All substrings were of the same length, and each was reverse complemented with probability 1/2. With the introduction of error, this collection of substrings constituted our input fragments.

To introduce error at rate ε into a substring of length n, we performed i insertions, d deletions, and s substitutions, by repeatedly selecting an insertion, deletion, or substitution at random while satisfying

$$i + d + s \leq \varepsilon(n + i - d).$$

Since the relationship between the error in the input and the error in the output was of interest, we wanted to keep the edit distance between overlapping fragments as close as possible to the input error rate. Care was taken to ensure that any character was edited at most once when generating the input.

Table 2. Experiment parameters.

Group	Experiment	Sequence type	Sequence length	Fragment length	Number of fragments	Genome equivalents	Error rate (%)
1	1	Random	50,000	500	500	5	2.5
	2						5.0
	3						10.0
2	4	Human	50,000	500	500	5	0.0
	5						2.5
	6						5.0
	7						10.0
3	8	Human	50,000	250	1,000	5	2.5
	9						5.0
4	10	Human	50,000	1,000	250	5	2.5
	11						5.0
5	12	Human	73,360	1,000	367	5	2.5

The input parameters for the experiments were the sampled sequence, the error rate, the number of fragments, and the substring length, which from now on we call the *fragment length.* Since we recorded the positions of the substrings that were sampled, we knew the true layout of the fragments, and the correct reconstructed sequence, and could compare this with the output of our software. The software in addition required a match significance and error distribution threshold for overlap graph construction, and a window width for multiple sequence alignment. For the experiments, we used a match significance threshold of 10, an error distribution threshold of 5%, and a window width of 4.

Twelve experiments were performed in all, and their parameters are given in Table 2. For the first group of experiments, numbered 1–3, we generated a random sequence of length 50,000, with characters drawn uniformly and independently from the alphabet {a, c, g, t}. We then sampled this sequence with 500 fragments of length 500, a sample of roughly 250,000 characters. Sample size divided by sequence length is known as the number of *genome equivalents*, which was held at 5, for all experiments. These values (500 fragments of length 500, and 5 genome equivalents) are intended to reflect laboratory practice. The error rates we chose (2.5%, 5%, and 10%) far exceed those encountered in practice, which are often quoted at less than 1%. Our intention here was to explore the robustness of our approach to error in the data.

A random sequence has no structure, while biological sequences contain repeats. In the remaining experiments, numbered 4–12, we used the human β-like globin gene cluster sequence [23]. This 73,360-character sequence contains many approximate repeats, and presents a challenging reconstruction problem. Thirteen short interspersed *Alu repeats* are present, nine in the foward direction and four in the reverse, as well as eight long interspersed *L1Hs repeats*, of which two are forward and six reversed. The Alu repeats are well separated, and each is roughly 300 nucleotides long. The L1Hs repeats are recursive in structure, and contain as many as 2000 nucleotides. In addition, the sequence contains many exact repeats of 10–15 nucleotides.

It is our understanding that this degree of repetition is unusual. We chose the sequence because instances of this difficulty apparently arise, and we were interested in testing the limits of our approach. Experiments 4–11 took the first 50,000 characters of the human gene cluster sequence. The first 50,000 nucleotides contain all but two of the Alu repeats and one of the L1Hs repeats.

We point out that, within groups, fragments were formed from the same collection of substrings; only the error rate varied. Moreover, between the first and second groups, the position of the substrings, as well as the location of errors, was the same; only the underlying sequence varied. Thus, any difference in output between Groups 1 and 2 is due to the structure of the sequence, rather than the pattern of sampling.

The effect of fragment length was examined in the third and fourth groups. Motivated by the results in Group 4, we decided to perform Experiment 12, in which the entire gene cluster sequence was sampled at 5 genome equivalents, by fragments of length 1000.

6.1. Synopsis of Results. The experiments may be divided into Experiments 1–3 on the random sequence, and Experiments 4–12 on the biological sequence. The point to keep in mind is that the first set of experiments contains essentially no repeats, while the second set contains many. On the random sequence experiments, the exact algorithm found layouts that were provably optimal, as the maximum-weight branchings were already dovetail-chain. On the biological sequence experiments, however, the exact algorithms could not solve Experiments 4–7 to optimality. Moreover, the greedy algorithms found the optimal solutions of the exact algorithms on Experiments 1–3, and found solutions equivalent to the exact algorithms on Experiments 4–7. In light of this, on Experiments 8–12 we ran only the greedy algorithms, by limiting the search with $K = 1$.

The lesson we draw from this experience is that the exact algorithms work well in the absence of repeats, but for a sequence as repetitive as the human gene cluster sequence, they are incapable of finding an optimal solution. Moreover, the greedy algorithms appear to work just as well, and produce layouts of acceptable quality.

We now present detailed results with respect to various performance measures. Those of primary interest are the quality of the layout and consensus sequence, and we report these first. Layout statistics are for the greedy algorithms, as just explained. We follow with some interesting parameters of the overlap graphs and multiple sequence alignments, report computation times, and describe how the software was run.

6.2. Layout Quality. Four measures of layout quality are summarized in Table 3. The first measure, *number of contigs*, is expressed as a composite number $x/y/z$. Here x is the number of contigs in the computed layout, y is the number in which every fragment was correctly ordered, and z is the number of contigs in the correct layout, given that some edges were erroneously culled from the overlap graph. In other words, z is the number of contigs in a perfect reconstruction that is restricted to the overlaps in the graph. In all experiments, the number of contigs in the true layout is z minus the number of incorrect culls.

Table 3. Layout quality.

Experiment	Fragment length	Number of fragments	Error rate (%)	Number of contigs[a]	Incorrect culls	Incorrect adjacencies[b]	Overlap savings
1	500	500	2.5	6/6/6	0	0/494	0
2			5.0	6/6/6	0	0/494	0
3			10.0	8/8/8	2	0/492	0
4	500	500	0.0	7/6/6	0	9/494	268
5			2.5	7/6/6	0	16/494	767
6			5.0	7/5/6	0	19/494	1076
7			10.0	10/9/8	2	27/492	994
8	250	1000	2.5	8/7/8	1	21/992	824
9			5.0	8/6/8	1	23/992	1110
10	1000	250	2.5	4/2/3	1	7/247	−56
11			5.0	3/1/2	0	10/248	2164
12	1000	367	2.5	6/5/5	0	5/362	784

[a] The number of contigs $x/y/z$ is expressed as the number x in the computed layout, followed by the number y in which every fragment is correctly ordered, followed by the number z in the correct layout given that some overlaps were culled.
[b] The number of incorrect adjacencies x/y is expressed as the maximum x of the number of pairs of fragments adjacent in the computed layout order, or the correct order, but not both, followed by the total number y of pairs in the correct order.

All *incorrect culls*, the second measure, occurred because the score of an overlap was below the overlap threshold. To give an example, the one incorrect cull in Experiments 8 and 9 occurred because the substrings involved had an overlap in the true layout of only 13 characters. With deletion errors, the score for this overlap was less than the threshold of 10, causing it to be culled during overlap graph construction. While the true layout consisted of seven contigs, without this overlap it broke into eight. To permit a fair comparison with the computed layout—which can be formed only on the basis of overlaps in the graph—we report the number of contigs in the correct layout as eight for these two experiments, and give in a separate column the number of incorrect culls. Correct layout, from now on, means the true layout given incorrect culls.

The third measure is the number of *incorrect adjacencies*. Counting contigs that are completely correct is a coarse measure of quality. For example, while only six of the seven contigs computed in Experiment 4 were completely correct, much of the seventh contig was correct as well. We took as a measure of the degree of correctness, the number of pairs of fragments that were adjacent in layout order in both the computed layout and the correct layout, where fragments are ordered first by increasing left endpoint, and second by decreasing right endpoint. To count the number of incorrect adjacencies, we tallied the number of pairs adjacent in one order but not the other, and took the maximum of the tallies for the two orders. In Table 3, incorrect adjacencies are expressed as this count, followed by the total number of adjacencies in the correct layout.

As an example, consider a layout of six fragements in two contigs, correctly ordered $\langle 1, 2, 3\rangle$ and $\langle 4, 5, 6\rangle$. Suppose we take fragment 5 and move it after fragment 3, breaking the layout into ⟨1, 2, 3, 5⟩, ⟨4⟩, and ⟨6⟩. This layout has one pair, (3, 5), which is not present in the original order, while the correct layout has two pairs, (4, 5) and (5, 6), not present in the incorrect order. We count this as two incorrect adjacencies.

The last measure is *overlap savings*. This is the weight of the branching inducing the computed layout, minus the weight of the branching over the same overlap graph inducing the correct layout. A positive quantity means that the computed layout has greater overlap, or is in a sense shorter.

The most striking feature of Table 3 is that all the random sequence experiments, and none of the biological sequence experiments, were solved correctly. This suggests that the exact and greedy algorithms work well in the absence of repeats. In the presence of repeats, the greedy algorithm found a layout shorter than the correct one for all experiments except Experiment 10, and, at error rates of 5% or less, it correctly determined over 95% of the adjacencies. Perhaps with the methods of Section 4.6 for producing alternate layouts and accommodating layout constraints, a biologist could correct the remaining 5%.

As a general trend within a group, the compression and rearrangement within layouts increased at higher error rates. This can be explained by approximate repeats in the gene cluster sequence, since our criterion of minimizing layout length will compress approximate repeats, assuming they are long and occur at a low error rate. Comparing the second, third, and fourth groups at the same error rate, the number of incorrect adjacencies increased with the number of fragments in absolute terms, but as a fraction of the total number of adjacencies, there is no discernible relation. This is probably due to variation in the pattern of repeats that were sampled when the fragment length varied across groups.

One statistic not presented in Table 3 is the number of incorrect orientations. This is because, for all twelve experiments, the relative orientation of fragments was correctly determined in all contigs, even when a contig contained incorrectly placed fragments. This is somewhat surprising, since, not counting the experiments within a group at varying error rates, there were more than 2500 orientations to determine, of which roughly half were reverse complements. Either the internal reverse complementarity of the gene cluster sequence is sufficiently simple, though reverse complement repeats are present, or fragment orientation is easier than fragment layout.

6.3. Consensus Error. Measures of the error in the reconstructed sequence are given in Table 4. These also require some explanation.

We examined the consensus sequence in Experiments 5–7 for two contigs that were correctly laid out in all three experiments. These consensus sequences were compared with the correct sequence by counting the number of insertion, deletion, and substitution errors. The raw error is given in the column, *in sample*, under *tally*, as the error count followed by the number of characters in the sample. The next column expresses the error as a rate, for example, 1 error in 560 characters.

Most of the errors in the consensus occurred when only one or two fragments participated in the voting. When only one fragment votes, neither the presence nor absence of an error can be detected; when two vote, an error can be detected,

Table 4. Consensus error.

		Output error[a]					
		In sample		At coverage ≥ 3		With correction	
Experiment	Input error rate	Tally	Rate	Tally	Rate	Tally	Rate
5	1 in 40	11/6163	1 in 560	5/5379	1 in 1075	0/5374	0 in 5374
6	1 in 20	29/6158	1 in 212	3/5367	1 in 1789	1/5365	1 in 5365
7	1 in 10	41/6154	1 in 150	9/5422	1 in 602	0/5413	0 in 5413

[a] The output error tally x/y is expressed as the number x of insertion, deletion, and substitution errors between the computed consensus sequence and the correct sequence, followed by the number y of characters in the consensus sequence, over a sample of the output.

but not corrected. Consequently, counting errors at a coverage of one or two fragments misrepresents the error rate, since few users would accept such a consensus. The column, *at coverage* ≥ 3, gives the error in the sample when three or more fragments participated in the voting.

Of the 17 errors at coverage at least three, all but one were insertion errors caused by the configuration of Figure 9. This configuration is characterized by two adjacent columns, where the first half of the rows contains a−, the middle row contains aa, and the last half contains −a. Of course, the choice of character a is arbitrary, as well as the order of the two columns, and the order of the rows.

Suppose both halves contain at least $\lceil k/2 \rceil - 1$ rows, where k counts the total number of rows. Since each column then contains at least $\lceil k/2 \rceil$ characters, the voting procedure of Section 5 will interpret the alignment as the result of $k - 1$ deletions, and correct them by outputting AA for the consensus. Consider sliding the characters of the top half one column to the right; this also gives a valid alignment. Our alignment algorithm of Section 5, which approximates TRACE, may not distinguish between these two configurations, or may even prefer the first configuration to the second. Our consensus algorithm, on the other hand, will interpret the second configuration as the result of one insertion, and output a single A. This is a more parsimonious explanation of the data, and in our experiments, it was the correct one.

The configuration of Figure 9 is not hard to recognize and correct prior to

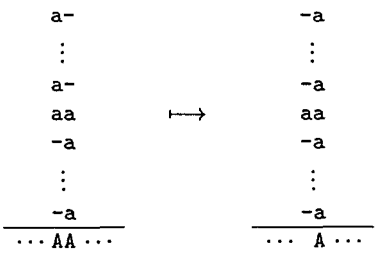

Fig. 9. A typical configuration causing an error in the consensus, and its correction.

voting. The column, *with correction*, gives the error in the sample at coverage at least three, taking this into account.

It is clear from the table that the error in the reconstructed sequence, at a coverage of three or more, is already less than 1 in 1000 for input rates of 5% or less. With the correction described above, it is less than 1 in 5000. Of course, we sampled little more than 5000 characters, so this statistic may be inaccurate.

Nevertheless, 1 in 5000 is a dramatic improvement over an error of 1 in 20. These experiments were at an average coverage of five (see Section 6.4), which indicates rapid convergence to the underlying sequence. Perhaps a coverage of eight would yield output of sufficient accuracy for most conceivable applications.

6.4. Coverage Depth and Vertex Degree. In Sections 4 and 5 we claimed that, in practice, the coverage depth of a layout, and the vertex degree in an overlap graph, are both small constants relative to the number of fragments. Table 5 presents some statistics for these two parameters. *Average coverage depth* was computed by summing the number of spanned sequences over all columns of the multiple sequence alignment, and dividing by the number of columns. *Average vertex degree* was computed by dividing the number of edges in the overlap graph by the number of vertices. *Maximum vertex degree* was computed by counting the number of in-edges and out-edges of each vertex, and taking the maximum of the two. (Taking the sum would yield a figure that should be compared with twice the average degree.)

The data supports our assumption that both the expected coverage depth and vertex degree are near the number of genome equivalents, which in practice is a

Table 5. Coverage depth and vertex degree.

				Coverage depth[a]		Vertex degree[b]	
Experiment	Number of fragments	Error rate (%)	Genome equivalents	Average	Maximum	Average	Maximum
1	500	2.5	5	5.1	14	5.0	17
2		5.0		5.1	14	5.0	16
3		10.0		5.2	14	5.2	17
4	500	0.0	5	5.1	14	5.2	17
5		2.5		5.2	15	5.3	17
6		5.0		5.2	15	5.6	17
7		10.0		5.3	16	6.3	19
8	1000	2.5	5	5.2	19	5.4	30
9		5.0		5.2	19	5.6	30
10	250	2.5	5	5.1	12	5.2	14
11		5.0		5.2	12	5.4	15
12	367	2.5	5	5.2	13	5.3	14

[a] Average coverage depth is the sum, over all columns of the alignment, of the number of spanned sequences divided by the total number of columns.

[b] Average vertex degree is the number of edges in the overlap graph divided by the number of vertices. Maximum vertex degree is the maximum number of in-edges or out-edges for any vertex.

constant. The maximum values are much higher, but still more than an order of magnitude less than the number of fragments.

6.5. Computation Time. Our software took as input the fragments, the error rate, and some additional parameters. These parameters were the overlap graph thresholds, the multiple alignment window-width, and the maximum search-tree size. As mentioned earlier, for all experiments we used a match significance threshold of 10, an error distribution threshold of 5%, and a window width of 4. For the search trees, we initially tried an unbounded number of nodes for fragment orientation, and an unbounded number of generated branchings for fragment layout. Optimal solutions were found on Experiments 1–3, the random sequence group; the maximum-weight branchings for these experiments were dovetail-chain. We were unable, however, to find an optimal orientation or layout for Experiments 4–7 within overnight runs. Hence, we decided to forego branch-and-bound and use greedy extension for fragment orientation, and greedy repair for fragment layout, on Experiments 4–12. One greedy orientation was computed, and one maximum-weight branching was generated and repaired. As all fragments were correctly oriented, and all layouts except Experiment 10 had an amount of overlap at or exceeding that of the correct solution, it appears that the greedy algorithms perform surprisingly well.

Table 6 gives computation times. *Overlap time* is the time for overlap graph construction, *layout time* is the time for fragment orientation and layout by the greedy algorithms (the algorithms that produced layouts for Table 3), and *alignment time* is the time for multiple alignment and consensus voting. Times are in units of hours, seconds, and minutes, respectively, on a six-processor Silicon Graphics Iris 4D/300GTX, running at 33 MHz, with 16 Mbytes of RAM. Our code did not exploit any parallelism in the machine.

As Table 6 shows, nearly all the running time was taken by overlap graph construction. The data supports a linear growth in overlap time as a function of

Table 6. Computation time.

Experiment	Fragment length	Number of fragments	Error rate (%)	Overlap time (hr)	Layout time (s)	Alignment time (min)
1	500	500	2.5	0.9	6	2.9
2			5.0	1.8	6	2.9
3			10.0	2.5	7	3.0
4	500	500	0.0	0.4	4	2.8
5			2.5	1.1	6	3.0
6			5.0	1.8	6	2.8
7			10.0	3.3	8	3.0
8	250	1000	2.5	1.3	15	4.3
9			5.0	1.8	16	4.4
10	1000	250	2.5	1.2	2	2.5
11			5.0	2.4	3	2.5
12	1000	367	2.5	2.5	5	4.5

error rate, which agrees with the worst-case time bound of $O(\varepsilon N^2)$ for fragments of total length N at error rate ε. Experiments 1–11 all have roughly the same number of input characters, so another study would be necessary to observe a quadratic growth in input length.

We point out that while overlap time is on the order of hours, it may be amortized over the period of data acquisition. As the sequence of each fragment is obtained, the overlap graph can be updated with the insertion of one vertex, and the comparison of the new fragment to those currently in the graph. If we divide the time to compute the overlap graph by the number of fragments, the amortized time is less than 10 s per fragment for 500 fragments of length 500 at 2.5% error. For 250 fragments of length 1000 at 2.5% error, the time is less than 5 s, though the response time will of course degrade for the last fragments inserted. Note that the time to compute a layout is also on the order of 5 and 10 s. This suggests that, for problems of current size, it is possible to deliver an updated layout as each fragment is sequenced. And as Section 6.2 indicates, the layout may be of acceptable quality even for problematic repetitive sequences.

7. Conclusion.

A four-phase algorithm for sequence reconstruction has been presented. For a problem involving V fragments of total length N, the first phase constructs a graph of overlaps within error rate ε in time $O(\varepsilon N^2)$. Generally, the length L of the underlying sequence is known approximately, and experimentalists sample fragments until $D = N/L$ is between 2 and 10. Ratio D is the number of genome equivalents sampled, and is always a small constant. In such a case, the expected out-degree of a vertex in the overlap graph is $O(D)$, implying that the number of edges is in expectation $O(V)$. Orientation and layout phases then take time $O(KV \log V)$, where K is the size of their search space. Under most conditions, convergence is quick, and performance in practice is basically $O(V \log V)$. For the final multiple alignment phase, $O(D^2L)$ time is taken. Thus, under empirical conditions, the algorithm runs in roughly $O(\varepsilon N^2 + V \log V + D^2L)$ time.

The most time-consuming aspect of the computation in practice is the comparison of fragments during overlap graph construction. (Ironically, the only phase that is not NP-complete.) Current work focuses on trying to lever recent methods for fast database searching to produce a subquadratic algorithm for this phase.

Another weakness of our method is that it artificially separates orientation and layout. (See [18] for an explanation of why this is necessary, given our choice of a relaxation to maximum-weight branchings.) As we have noted, solving each of these problems optimally does not guarantee an optimal solution to the combined reconstruction problem. What is desired is an algorithm that solves both simultaneously. This can be done with a graph-theoretic formulation that uses a relaxation to maximum-weight matchings, which is the subject of a future paper.

Acknowledgments. The first author wishes to thank Webb Miller for providing the human gene cluster sequence, and David Sankoff for access to machines while at the Université de Montréal. Our thanks also to the anonymous referees for their comments.

References

[1] Blum, A., T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *Proceedings of the 23rd ACM Symposium on Theory of Computation*, pp. 328–336, 1991.

[2] Camerini, P., L. Fratta, and F. Maffioli. A note on finding optimum branchings. *Networks* **9**, 309–312, 1979.

[3] Camerini, P., L. Fratta, and F. Maffioli. The *k* best spanning arborescences of a network. *Networks* **10**, 91–110, 1980.

[4] Chang, W. and E. Lawler. Approximate string matching in sublinear expected time. *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*, pp. 118–124, 1990. To appear in *Algorithmica.*

[5] Chvátal, V., and D. Sankoff. Longest common subsequences of two random sequences. *Journal of Applied Probability* **12**, 306–315, 1975.

[6] Cull, P. and J. Holloway. Reconstructing sequences from shotgun data. In *Sequences II: Methods in Communication, Security, and Computer Science*, R. Capocelli, A. De Santis, and U. Vaccaro, eds., Springer-Verlag, New York, pp. 166–188, 1993.

[7] Foulser, D. A linear time algorithm for DNA sequencing. Technical Report 812, Department of Computer Science, Yale University, New Haven, CT 06520, 1990.

[8] Fredman, M., R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica* **1**, 111–129, 1986.

[9] Fredman, M., and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the Association for Computing Machinery* **34**(3), 596–615, 1987.

[10] Gabow, H. Two algorithms for generating weighted spanning trees in order. *SIAM Journal on Computing* **6**(2), 139–150, 1977.

[11] Gabow, H., Z. Galil, T. Spencer, and R. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica* **6**, 109–122, 1986.

[12] Gallant, J. The complexity of the overlap method for sequencing biopolymers. *Journal of Theoretical Biology* **101**, 1–17, 1983.

[13] Gallant, J., D. Maier, and J. Storer. On finding minimal length superstrings. *Journal of Computer and System Sciences* **20**(1), 50–58, 1980.

[14] Gingeras, T., J. Milazzo, D. Sciaky, and R. Roberts. Computer programs for the assembly of DNA sequences. *Nucleic Acids Research* 7(2), 529–545, 1979.

[15] Gusfield, D., G. Landau, and B. Schieber. An efficient algorithm for the all pairs suffix–prefix problem. *Information Processing Letters* **41**, 181–185, 1992.

[16] Huang, X. A contig assembly program based on sensitive detection of fragment overlaps. *Genomics* **14**, 18–25, 1992.

[17] Hutchinson, G. Evaluation of polymer sequence fragments data using graph theory. *Bulletin of Mathematical Biophysics* **31**, 541–562, 1969.

[18] Kececioglu, J. Exact and approximation algorithms for DNA sequence reconstruction. Ph.D. dissertation, Technical Report 91-26, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, 1991.

[19] Kececioglu, J., and E. Myers. A procedural interface for a fragment assembly tool. Technical Report 89-5, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, 1989.

[20] Lawler, E. A procedure for computing the *k* best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science* **18**, 401–405, 1972.

[21] Li, M. Towards a DNA sequencing theory. *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*, pp. 125–134, 1990.

[22] Manber, U. and G. Myers. Suffix arrays: A new method for on-line string searches. *Proceedings of the 1st Annual ACM–SIAM Symposium on Discrete Algorithms*, pp. 319–327, 1990. To appear in *SIAM Journal on Computing.*

[23] Margot, J., G. W. Demers, and R. Hardison. Complete nucleotide sequence of the rabbit β-like globin gene cluster: analysis of intergenic sequences and comparison with the human β-like globin gene cluster. *Journal of Molecular Biology* **205**, 15–40, 1989.

[24] Mehlhorn, K. *Data Structures and Algorithms*, Vol. 1. Springer-Verlag, Berlin, 1984.

[25] Myers, E. Incremental alignment algorithms and their applications. Technical Report 86-2, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, 1986.

[26] Peltola, H., H. Söderlund, J. Tarhio, and E. Ukkonen. Algorithms for some string matching problems arising in molecular genetics. *Proceedings of the 9th IFIP World Computer Congress*, pp. 59–64, 1983.

[27] Peltola, H., H. Söderlund, and E. Ukkonen. SEQAID: a DNA sequence assembly program based on a mathematical model. *Nucleic Acids Research* **12**(1), 307–321, 1984.

[28] Press, W., B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, 1988.

[29] Sankoff, D. Minimal mutation trees of sequences. *SIAM Journal on Applied Mathematics* **28**(1), 35–42, 1975.

[30] Sankoff, D. and V. Chvátal. An upper bound technique for lengths of common subsequences. In *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence comparison*, D. Sankoff and J. Kruskal, eds., Addison-Wesley, Reading, MA, pp. 353–357, 1983.

[31] Sankoff, D. and J. Kruskal, eds. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, MA, 1983.

[32] Shapiro, M. An algorithm for reconstructing protein and RNA sequences. *Journal of the Association for Computing Machinery* **14**, 720–731, 1967.

[33] Smetanič, Y., and R. Polozov. On the algorithms for determining the primary structure of biopolymers. *Bulletin of Mathematical Biology* **41**, 1–20, 1979.

[34] Smith, T. F., and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology* **147**, 195–197, 1981.

[35] Staden, R. A strategy of DNA sequencing employing computer programs. *Nucleic Acids Research* **6**(7), 2601–2610, 1979.

[36] Tarhio, J. and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science* **57**, 131–145, 1988.

[37] Tarjan, R. Finding optimum branchings. *Networks* **7**, 25–35, 1977.

[38] Turner, J. Approximation algorithms for the shortest common superstring problem. *Information and Computation* **83**, 1–20, 1989.

[39] Ukkonen, E. A linear algorithm for finding approximate shortest common superstrings. *Algorithmica* **5**, 313–323, 1990.