FAT - A Fragment Assembly Toolkit

João Meidanis
Computer Science Dept.
Univ. of Campinas
Brazil

October 2, 1995

Abstract

We describe a system for DNA fragment assembly. The system is composed of several programs, but here we concentrate on the preprocessing tool, the post-processing tool, and the main assembly module. We outline the algorithms and data structures used and present experimental results. Finally, we discuss possible extensions to the system.

1 Introduction

A DNA fragment assembly program receives a collection of DNA sequences, called fragments, and outputs a collection of (much) longer sequences, the *contigs*, that contain every fragment or its reverse complement as an approximate substring. Ideally, one single contig should be returned, but errors in the input data, lack of coverage, and other factors may prevent complete assembly.

It is conceivable that some systems can handle additional information — such as the approximate total length of the target DNA molecule being sequenced, or relative orientations between some fragments — to help the assembly process. Our system, however, is not prepared for this possibility. It relies solely in the overlap relationship of the fragments to reach its conclusions.

Our goal in building this toolkit is to provide a system that performs a good assembly very quickly. We feel that the computational problems involved can be quickly solved by modern computers. We are aware that many formlaizations of the fragment assembly problem lead to NP-hard problems. However, it is not clear that these formalizations are the only ones possible and the existence of a good formalization that leads to a polynomial time algorithm cannot be ruled out.

Some of the interesting features of the system are:

- a powerful technique that can quickly recognize pairs of fragments with high overlap (pre-processing tool)
- a novel and fast way of aligning fragments without resorting to a full dynamic progrmming algorithm
- a simple and efficient algorithm to compute multiple alignments from a multi-sequence trace (post-processing tool)

The rest of this paper is organized as follows. Section 2 describes the main assembly module. Section 3 talks about on the pre- and post-processing tools. Section 4 contains some results from experiments we conducted on data provided for the DIMACS Implementation Challenge. Finally, Section 5 has concluding remarks and future plans.

2 Assembly

Assembly operation is depicted schematically in Figure 1.

The program `assemble` receives as input a list of *fragment pairs* which were reported as probably significant for the assembly by the pre-processing phase. More specifically, each pair in the list is composed of two fragments that supposedly have prefix-suffix similarity, or containment similarity. Prefix-suffix similarity means that a prefix of one of the sequences is similar to the suffix of the other. Containment similarity means that one of the sequences is an approximate substring of the other. In addition, two other pieces of information are conveyed along with the pair. These are (1) an estimate of the *number of bases* by which they overlap, and (2) a value that indicates relative orientation of the fragments ($+1$ if they overlap in the orientation they are given, or -1 if they overlap when one of them is reversed and complemented).

The output of `assemble` is a list of contigs. Each contig is composed of a set of fragments, a multi-sequence trace involving the sequences of the fragments in the contig, and relative orientations of these fragments. The

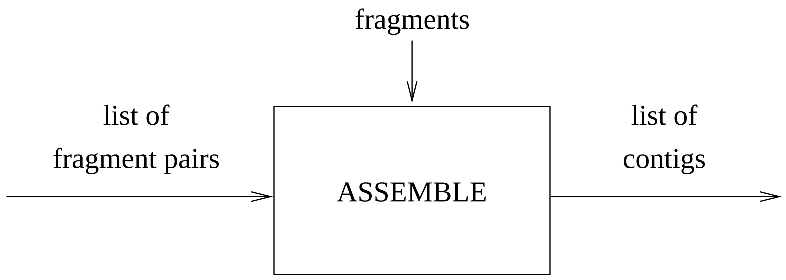


Figure 1: The assembly process.

fragment sets of different contigs are disjoint and their union is the whole fragment set given as input.

The program operates as follows. Initially, each fragment is made into a contig. Thus, we have as many contigs as fragments. The list of fragment pairs is then processed, in descending order of estimated overlap. For each pair, we align the fragments using a heuristic, in the reported orientations, and the quality of this alignment is assessed. If the alignment is acceptable, then we check whether the fragments already belong to the same contig or not. If they do, their relative orientation in the contig is compared to their relative orientation in the alignment, and a warning is issued when there is orientation incompatibility. However, the process continues since the last alignment is probably weaker. If the fragments belong to different contigs, then these contigs are merged into one bigger contig.

This checking and merging process continues until either the list of pairs is exhausted, in which case the remaining contigs are reported, or until only one contig remains.

2.1 Pair validation

We will now describe in more detail the verification step, where a supposedly related pair is checked. This step was recently improved to use a technique significantly faster than full dynamic programming to find good alignments between fragments. Before getting into that, we will describe the general processing of a pair.

First, the second fragment is reversed and complemented if the reported relative orientation is -1. Recall that this relative orientation is given along

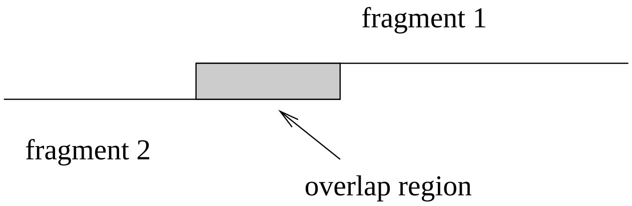


Figure 2: Overlap region.

with the pair by the pre-processor. Then, the best possible prefix-suffix or containment alignment is obtained between the two sequences. This alignment has an *overlap length* and an *score* relative to this overlap region (see Figure 2).

The alignment is considered *acceptable* if

$$\textit{overlap-length} \geq \texttt{MINOVLP}$$

and

$$\frac{\textit{score}}{\textit{overlap-length}} \geq \texttt{MINUSCORE}.$$

The parameters `MINOVLP` and `MINUSCORE` are set by the user. Default values of 12 and 0.85 are provided. The score is computed based on a standard scoring scheme involving matches, mismatches, and gaps in the overlap region as follows.

$$\textit{score} = \sum_{i=1}^{\textit{overlap-length}} s(a_i, b_i),$$

where a_i and b_i are the symbols at position i in the overlap region of sequence 1 and 2, respectively, and the function s is given by

$$s(a,b) = \begin{cases} p(a,b) & \text{if } a \text{ and } b \text{ are characters} \\ -g & \text{if one of } a,\, b \text{ is a gap,} \end{cases}$$

where p and g are specified by the user. The function p is normalized so that

$$\max_{a \in \Sigma} p(a) = 1.$$

Thus, an alignment is accepted if its overlap region has at least a certain minimum length and its *unit score*, or average score per position, is at least a certain given value. In other words, the absolute score is at least a certain fraction of the maximum possible score for regions of the same length. For instance, if the overlap length is 30 and the default values are in effect, the score has to be at least $30 \times 0.85 = 25.5$ for an alignment to be accepted.

When the alignment is accepted and the two fragments belong to distinct contigs, the contigs are merged. This involves the complete reversal of one of the contigs at times. In this case, the smaller contig (in number of fragments) is reversed. Another consequence of contig merging is the *unification* of matching bases in the accepted alignment. This is explained in more detail in Section 3.2.

2.2 Computing the best alignment

Originally, a standard dynamic programming (dp) algorithm was being applied to the two fragments involved. Prior to this procedure, the second fragment was reversed and complemented in case the relative orientation reported was -1. The dp algorithm was configured so that end gaps in either sequence are not penalized. This way, it returns good prefix-suffix or containment alignments.

However, this procedure has time complexity $O(mn)$, where m and n are the lengths of the fragments being compared. In practice, $m \cong n \cong 500$ characters. This time complexity was a problem. Typical assemblies involve 10^4 to 10^5 fragment pairs, even after the filtering done by pre-processing. Thus, it was difficult even to test the system with actual data from large-scale sequencing projects.

Recently, this part of the processing has been replaced by a faster heuristic method. Instead of applying dp on the whole fragments, *common windows* of length at least `WINDOW` are sought. The parameter `WINDOW` is set by the user, and defaults to 12. The best collection of compatible windows is then derived, and dp is applied only on the intervals between windows (see Figure 3).

A common window between two sequences is a common substring. A list of all common windows of length at least `WINDOW` can be efficiently computed using suffix trees [Apo85, GI93]. In our implementation, a related method, using the idea of suffix arrays [MM90], is employed. The minimum size requirement (parameter `WINDOW`) prevents the list of common windows from getting too large, which would defeat the overall goal of more speed.

After the list of common windows is constructed, a *combination* of them is sought to form the skeleton of a good alignment. This is similar to the

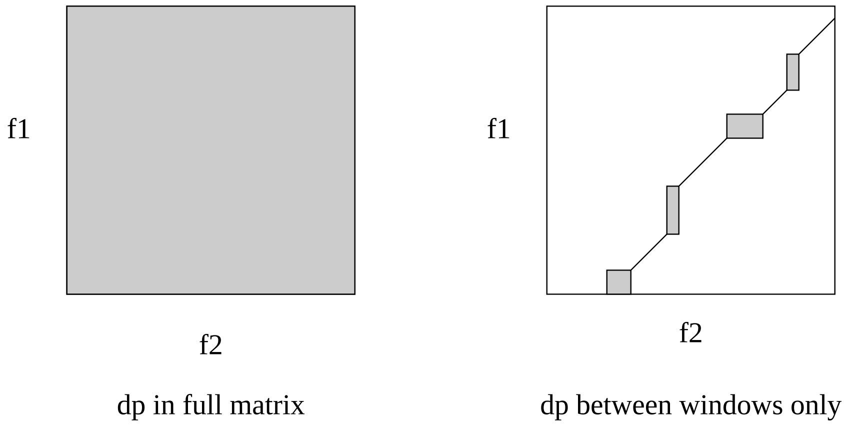


Figure 3: Windows approach.

approach followed in the well-known FASTA program [WL83, LP85, PL88, Pea90]. In our implementation, a method derived from [JMT92] is employed.

The main difference between our implementation and the method described in [JMT92] is the handling of intervals between windows. Given two common windows $[x^1_{\min}, x^1_{\max}] \times [y^1_{\min}, y^1_{\max}]$ and $[x^2_{\min}, x^2_{\max}] \times [y^2_{\min}, y^2_{\max}]$, the *interval* between them is the region $[x^1_{\max}+1, x^2_{\min}-1] \times [y^1_{\max}+1, y^2_{\min}-1]$. In the cited reference, those are given weight zero, while in the implementation we assign them a weight depending on their size. An $\Delta x \times \Delta y$ interval will be given weight

$$-g|\Delta x - \Delta y| + a \min(\Delta x, \Delta y),$$

where a is the average score for a base match. This weight reflects that there will be at least $|\Delta x - \Delta y|$ gaps in any alignment of the interval, plus the expected score for the aligned part, assuming all character matches equally probable.

In addition, common windows of length l are given weight l. We seek a combination of windows such that the total weight of the windows plus the intervals between them is maximized. This maximum is an approximation of the real score of an alignment based on these windows. From this combination, an actual alignment is obtained by matching the exact windows and applying a dp algorithm in the intervals.

3 Pre- and post-processing

In this section we briefly describe what is done in the pre- and post-processing phases.

3.1 Pre-processing

The pre-processing phase is aimed at finding quickly which pairs of fragments are likely to have a significant overlap.

Each pair of fragments is tested in all possible orientations. It suffices to perform two comparisons, keeping one of the fragments in a fixed orientation and trying the other one both as given and in reverse complement. This is because comparing f and g is equivalent to comparing f^{RC} and g^{RC}. Likewise, comparing f and g^{RC} is equivalent to comparing f^{RC} and g.

The comparison relies, again, on common windows. The idea is that, if two fragments overlap, then their sequences must have many common windows. The window size w is set by the user, and need not be the same used later in aligning the fragments. Its default value is 7.

One possible approach here would be to count the common windows between the two sequences. However, a faster method is used in our implementation, which is to *estimate* the number of common windows by statistical methods.

More precisely, we construct, for each sequence, the *set* of all windows of length w. Then, we compute a short *signature* of the set and compare these signatures instead of comparing the sets.

The signatures are computed as follows. Each word of length w over the alphabet $\{\mathtt{A}, \mathtt{G}, \mathtt{C}, \mathtt{T}\}$ is given a distinct number between 0 and $4^w - 1$. A set of windows of length w becomes then a set of integers from the range $[0, 4^w - 1]$. Instead of representing each set as a bit-vector of size 4^w, which could be too big, we *hash* the numbers from $[0, 4^w - 1]$ into a much smaller interval $[0, q-1]$, where q is a prime number. The hash function h is simply

$$h(x) = x \bmod q.$$

Hence, singatures are bit-vectors of size q. To estimate the number of common windows between sequences f and g we count the number of common bits in these vectors and apply a correction to compensate for the effect of hashing. Counting bits is astonishingly fast because one can use bit operations in entire words. A 2000-fragment set can be compared in this way (all against all) in about 8 minutes in a SPARC 10 or equivalent machine.

We end this section with some more detail on this correction formula, since we feel it is a method that can be used in other applications and readers can benefit from this extra information.

Let $\mathcal{W}_w(f)$ be the set of windows of length w of a sequence f. We are trying to estimate

$$a = |\mathcal{W}_w(f) \cap \mathcal{W}_w(g)|,$$

and we compute

$$c = |h(\mathcal{W}_w(f)) \cap h(\mathcal{W}_w(g))|.$$

This looks just like a classic statistical estimation problem in the following way. Let $A = \mathcal{W}_w(f)$ and $B = \mathcal{W}_w(g)$. Suppose we knew the values of a, $|A|$, and $|B|$. The we can compute the statistical distribution of c for A, B uniformly distributed, assuming h is "random". Standard occupancy models yield this distribution [JK77, KSC78, Gon81]. In particular, a formula for $\bar{c}$, the distribution mean, can be derived:

$$\bar{c} = |A|' + |B|' - (|A| + |B| - a)',$$

where x' is an abbreviation of

$$x' = q\left(1 - e^{-x/q}\right),$$

with q being the hash table size and e being the base of natural logarithms. Solving this equation for a, we can obtain an estimator $\hat{a}$ for a given c, $|A|$, $|B|$:

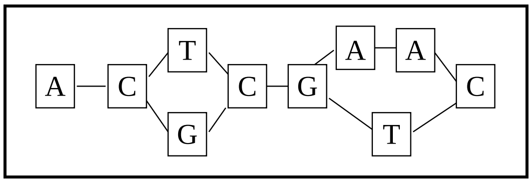
$$\hat{a} = |A| + |B| + q\ln\left(e^{-|A|/h} + e^{-|B|/h} - 1 + \frac{c}{q}\right).$$

More details on this method can be found in [MJ92, Mei92]. A related method is described in [DN82].

The method is very fast, but it has two drawbacks. First, since we are computing the *set* of common windows for a fragment, repetitions of the same window will be lost. Second, only errorless windows are counted. If a window has any character other than `A`, `G`, `C`, or `T`, the window is discarded. Thus, this method can perform poorly when there are repeats or a high error rate.

Alignment `ACTCGAAC`

`ACGCGT-C`



Unified structure

Figure 4: Unification.

3.2 Post-processing

One of the pieces of information stored along with each contig during assembly is a multi-sequence trace. These traces are internally represented as linked lists of bases, an idea already used by Hein [Hei90]. As contigs are merged, base unification occurs. Contigs are always merged based on an alignment between two fragments. Each column of the alignment that contains identical bases causes these bases to be unified in the trace structure. An example is given in Figure 4, where two singleton contigs (contigs with only one fragment) are merged based on the alignment shown.

The post-processing phase receives a contig and delivers a multiple alignment of the fragments and a consensus sequence.

We employ a greedy multiple alignment strategy described in detail elsewhere to construct the alignment given the trace [MS95]. From it, the consensus is obtained by a simple voting scheme. The consensus character in a column is the character with highest frequency in that column. If more than one character has maximum frequency in a column, one of these is chosen according to a fixed priority scheme.

4 Results

We ran our system with data provided by the organizers of the 1995 DIMACS Implementation Challenge. The data consists of several sets. Each

Set	Number of Fragments	Target Size	Coverage	Avg. size of Fragments
Seto, Koop, Hood				
247-ed	828	34476	11.4	475
Safer (GTC)				
b1496	1811	41516	10.2	233
b2126	1504	44566	7.8	232
l247	1704	44657	9,4	245
l518	1910	41631	10.4	227
HUMHBB				
c3	440	73308	3.0	500
c5	734	73308	5.0	500
c7	1027	73308	7.0	500

Table 1: The input fragment sets. NOTE: GTC SETS HAVE PROBLEMS: NEED CORRECTED CONSENSUS AND/OR FRAGMENTS

set contains a collection of fragments and a consensus sequence. The idea is to run the programs on the fragments, and then compare the resulting contigs to the consensus using a local alignment tool [HM91, CZOM95].

Table 1 shows the data sets and, for each set, its origin and several characteristics important for assembly.

The first set is from Seto, Koop, and Hood [SKH93]. It has a nonzero error rate, but less than 1% of the total bases are different from `A`, `G`, `C`, `T`.

The next four sets are from H. Safer. No characters different from `A`, `G`, `C`, `T` exist in these sets.

Finally, the last three sets are derived from the Human Beta Globin Gene Cluster, a 73 kbp sequence, obtained from Genbank under the code HUMHBB. These sets were constructed with the aid of the `genfrag` tool [EB93, EB94]. This tool generates random fragments from a given sequence. It produced the three sets listed last in Table 1 when configured to generate fragments with an average of 500 bases and a maximum deviation of 100 bases from the average, and mean coverage of 3, 5, and 7.

In addition, errors were introduced in the HUMHBB sets according to certain experimentally observed probabilities [CH92]. There probabilities are position-dependent inside the fragment, to reflect more closely what happens in practice. From each of the sets c3, c5, and c7 we derived three

Similarity range	Number of Contigs
90% – 100%	8
80% – 90%	10
70% – 80%	9
60% – 70%	4
50% – 60%	4

Table 2: Similarity results for 247-ed. NOTE: THESE RESULTS WILL BE SIGNIFICANTLY DIFFERENT IN FINAL VERSION

other sets incorporating different error rates. The probabilities of [CH92] yield an overall error rate of about 2.5%. These probabilities were multiplied by factors 1.0, 1.5, and 2.0 giving the final fragment sets which were fed into the assembly tool. Thus, we have a total of nine HUMHBB sets. The final sets are not shown in Table 1 because their characteristics, except from the error rate, are the same as those of their parent sets, which are listed.

The results are shown in several tables and figures that follow. Tables 2 and 3 and Figures 5 and 6 report the quality of the solutions while Table 4 shows the CPU time spent on each set.

The nature of the solution's quality is quite distinct depending on the set origin. Therefore, we present each kind in a separate way, which highlights the most interesting aspects of the solution.

The set 247-ed, from Seto, Koop, and Hood, has a very low coverage and an error rate of at least 1%. The solution given by our tools has 35 contigs. Table 2 has to do with the similarity between the contigs and the regions they cover in the consensus. For each similarity range 90% – 100%, 80% – 90%, etc. the table shows how many contigs were in that range.

The last four contigs, with similarity below 60%, could not be placed precisely on the map that follows, because the similarity was equally poor in both orientations, but on different locations.

We drew a map in Figure 5 to show the coverage and overlap pattern of the contigs. In this map, the consensus sequence is represented by a horizontal line at the bottom, and each contig is represented as a box whose projection on the consensus line identifies the region it covers. The height of the box is proportional to the number of fragments in the contig. All contigs in this set were very small, with a minimum of 1 and a maximum

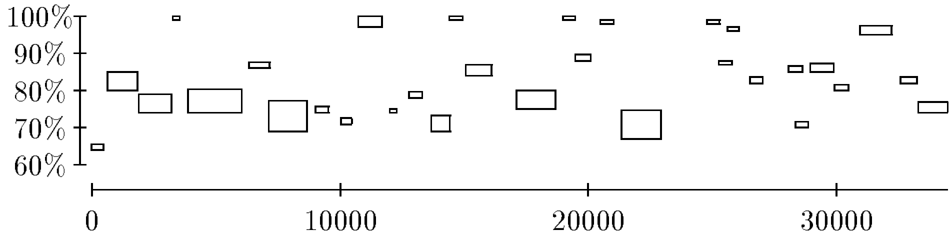


Figure 5: Map of resulting contigs for 247-ed.

of 12 fragments. The vertical position of a box is related to the precent of similarity the corresponding contig exhibited with the part of the consensus it covered, according to the scale on the right. This information is somewhat redundant with that of Table 2. Here the exact similarity, and not just the rate, can be seen. The vertical position of a box is the vertical position of its baseline (the bottom horizontal edge).

Table 3 shows the results concerning data made available by H. Safer of GTC. In these sets, the overall characteristics are high coverage and no errors. As a result, our system delivered very few contigs. The table shows the number of contigs generated for the sets, and, for each contig, its size, the percentage of consensus it covers, the number of fragments it contains, and the overlap (if any) with the previous contig in the list. As we can see, our toolkit was able to produce acceptable results in most of the sets in this class.

In two sets, our programs were not able to include a small fragment in the assembly. This is probably due to the inability of the pre-processor to find a high enough estimate of overlap with other fragments. However, in both cases the small fragment was just part of some already well covered region.

One thing that should be noticed is that a similarity level of 99.00% is not great. For sequences of the sizes involved here, the 1% that is missing represents about 400 bases, which is far from being negligible. The toolkit failed to get a better similarity level probably because of repeats present in the consensus.

(NOTE: THIS PART WILL HAVE TO BE REWRITTEN DUE TO CORRECTED CONSENSUS AND/OR FRAGMENTS OF GTC DATA)

Finally, Figure 6 shows the results concerning HUMHBB data. We were

Set	Number of Contigs	Contig Size (bp)	Percent Covered	Number of Fragments	Overlap w/ previous (bp)
b1496	2	42145	99.00%	1810	
		35	0.08%	1	35
b2126	3	5942	13.33%	178	
		28144	70.64%	1105	309
		7248	16.26%	221	
l247	1	38654	83.87%	1704	
l518	2	36391	99.20%	1909	
		23	0.06%	1	23

Table 3: Results for the GTC-Safer sets.

unable to run all the programs in the coverage 5 and coverage 7 data, so we only show coverage 3 results here. The figure contains results concerning three sets, all with the same coverage, but with different error rates.

These sets are characterized by a very long consensus, a significant error rate, and many repeats. HUMHBB is known to have several directed and inverted repeats in its sequence [Kec91].

As in the previous figure, the consensus is represented by a horizontal line in the bottom. Also, the contigs are boxes. However, unlike the previous figure, here the height of the box indicates the contig's *mean coverage*. This mean coverage is computed as

$$(\textit{avg. fragment length}) \times (\textit{number of fragments})/(\textit{contig size}),$$

where *contig size* is measured in base pairs (bp). This formula is not used for singleton contigs where the mean coverage is obviously 1.

In these data sets almost all contigs showed very high similarity with their covered consensus regions. Exceptions are represented by black boxes in Figure 6, and are listed below.

- in the err.1.0 set, there was one bad contig that exhibited 69% similarity. This is the larger contig, accouting for about 1/3 of the total fragments.

- in the err.2.0 set, two contigs exhibited similarities of 69% and 66%; these contigs have 18% and 7% of the fragments, respectively.

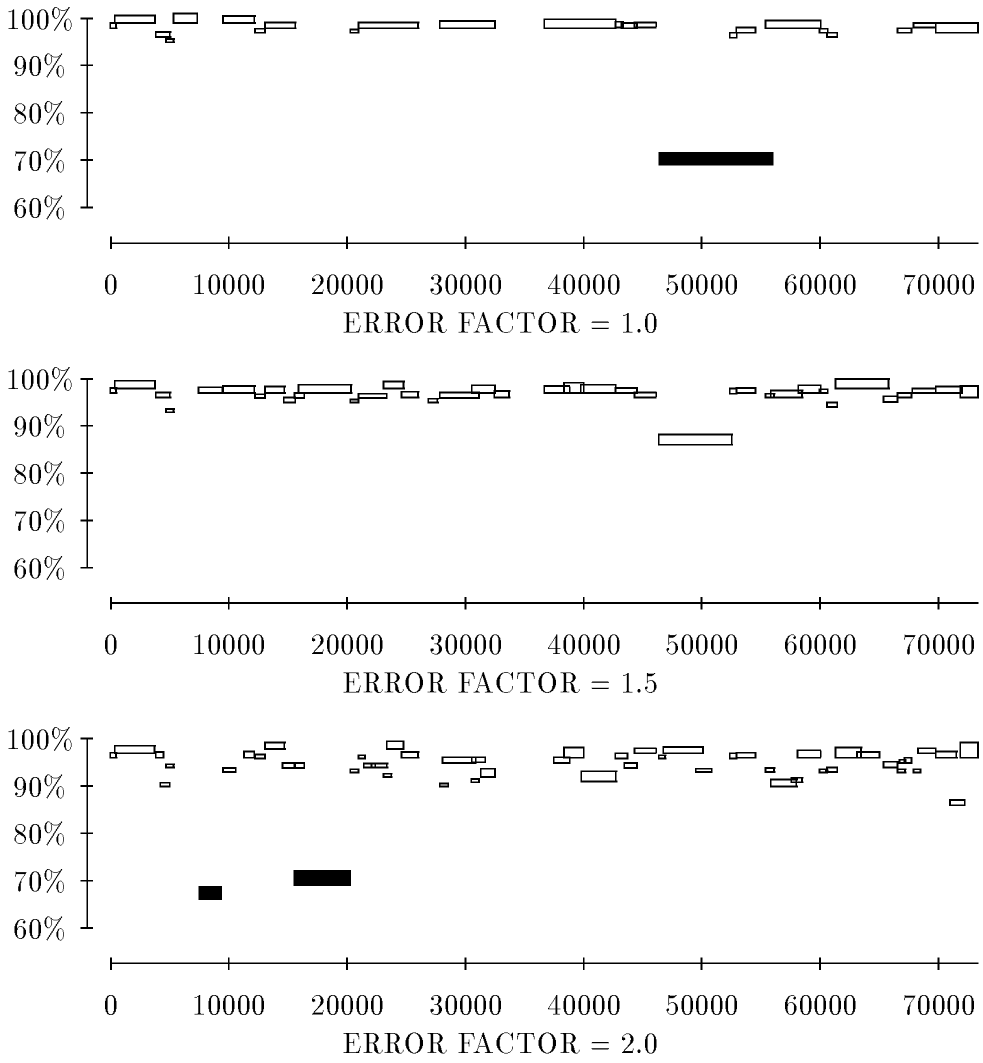


Figure 6: Maps of resulting contigs for HUMHBB, coverage 3.

Set	Pre-proc. (sec.)	Assembly (sec.)	Post-proc. (sec./contig)
247-ed	12	74	0
b1496	380	731	32
b2126	265	467	8
l247	344	657	31
l518	434	1489	39
cov 3, err 1.0	24	1585	0.8
cov 3, err 1.5	24	1332	0.2
cov 3, err 2.0	25	3317	0.0

Table 4: Processing times.

All other contigs exhibit similarity rates higher than 85% with their covered regions in the consensus. Notice that these contigs are short, so this similarity rate is good.

The overall results here show that our system tends to get confused when repeats are present. We analyzed the input data and verified that the coverage 3 set does not form a single contig. Actually, the fragments can be divided in 24 different contigs, with no fragment linking two of these contigs. Hence, any program that gets less that 24 contigs for this coverage 3 data must link fragments that it shouldn't. Our system got 24 contigs in the err.1.0 set, 38 in the err.1.5 set, and 54 in the err.2.0 set. This shows a conservative behavior, that is, fragments are only linked when we are sure that they have a good overlap. However, the presence of repeats was probably responsible for the poor similarity rates of the longer contigs in each case.

Finally, we show the processing times of each phase of the computation for each set in Table 4. The measurements were performed in SPARC 10 or equivalent workstations.

5 Conclusions

We have described the main features of a system to help fragment assembly in large-scale shotgun DNA sequencing projects. Experimental results for real data are shown.

The system is in general very fast. It took less than half an hour to process most fragment sets. However, in some cases it produced contigs that were only weakly similar to the true consensus. This happened mostly in the presence of repeats.

Planned additons to the system include new algorithms to deal with repeats and an improved memory management scheme. We also plan to integrate it in a wider system that supports scientific applications based on workflow [MVW95].

Acknowledgements

We thank Debby Joseph and Fred Blattner, who introduced us to fragment assembly, and our students A. A. M. Almeida, A. Moriya, D. Jerozolismki, and A. R. Cruvinel for help with the implementation. This work was partially supported by financing agencies FAPESP, CNPq, and FAEP-UNICAMP.

References

[Apo85] A. Apostolico. The myriad virtues of subword trees. In A. Apostilico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series F: Computer and System Sciences*, pages 85–96. Springer-Verlag, 1985.

[CH92] W.-Q. Chen and T. Hunkapiller. Sequencing accuracy of large dna sequencing projects. *DNA Sequence*, 2:335–342, 1992.

[CZOM95] K.-M. Chao, J. Zhang, J. Ostell, and W. Miller. A local alignment tool for very long dna sequences. *Computer Applications in the Biosciences (CABIOS)*, 11(2):147–153, 1995.

[DN82] J. Dumas and J. Ninio. Efficient algorithms for folding and comparing nucleic acid sequences. *Nucleic Acids Research*, 10(1):197–206, 1982.

[EB93] M.L. Engle and C. Burks. Artificially generated data sets for testing dna fragment assembly algorithms. *Genomics*, 16:286–288, 1993.

[EB94] M.L. Engle and C Burks. Genfrag 2.1: New features for more robust fragment assembly benchmarks. *Computer Applications in the Biosciences (CABIOS)*, 10:567–568, 1994.

[GI93] R. Grossi and G. F. Italiano. Suffix trees and their applications in string algorithms. In R. Baeza-Yates and N. Ziviani, editors, *Proc. 1st South American Workshop on String Processing*, pages 57–76, September 1993.

[Gon81] Gaston H. Gonnet. Expected length of the longest probe sequence in hash code searching. *JACM*, 28(2):289–304, April 1981.

[Hei90] Jotun Hein. Unified approach to alignment and phylogenies. In Russell F. Doolittle, editor, *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, volume 183 of *Methods in Enzymology*, pages 626–645. Academic Press, 1990.

[HM91] X. Huang and W. Miller. Linear-space local similarity algorithm. *Advances in Applied Mathematics*, 12:337–357, 1991.

[JK77] N. L. Johnson and S. Kotz. *Urn Models and Their Application.* John Wiley & Sons, New York, 1977.

[JMT92] D. Joseph, J. Meidanis, and P. Tiwari. Determining DNA sequence similarity using maximum independent set algorithms for interval graphs. In *Proc. of the Third Scandinavian Workshop on Algorithm Theory*, volume 621 of *Lecture Notes in Computer Science*, pages 326–337, 1992.

[Kec91] John D. Kececioglu. *Exact and approximation algorithms for DNA sequence reconstruntion.* PhD thesis, The University of Arizona, 1991.

[KSC78] Valentin F. Kolchin, Boris A. Sevast'yanov, and Vladimir P. Chistyakov. *Random Allocations.* Scripta Series in Mathematics. V. H. Winston & Sons, Washington, D.C., 1978.

[LP85] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity search. *Science*, 227:1435–1441, 1985.

[Mei92] J. Meidanis. *Algorithms for Problems in Computational Genetics.* PhD thesis, University of Wisconsin-Madison, 1992.

[MJ92] J. Meidanis and D. Joseph. Hashing and back prediction methods for computing sequence similarity for fragment assembly. Unpublished manuscript, 1992.

[MM90] U. Manber and G. Myers. Suffix arrays: A new method for on–line string searches. In *Proceedings of the First Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 319–327. SIAM, 1990.

[MS95] J. Meidanis and J. C. Setubal. Multiple alignment of biological sequences with gap flexibility. In R. Baeza-Yates, E. Goles, and P. V. Poblete, editors, *II Latin American Theoretical Informatics*, volume 911 of *LNCS*, pages 411–426, Valparaíso, Chile, April 1995. Springer-Verlag.

[MVW95] C. B. Medeiros, G. Vossen, and M. Weske. WASA: A workflow-based architecture to support scientific database applications. Technical Report 02/95-1, University of Münster, 1995.

[Pea90] William R. Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. In Russell F. Doolittle, editor, *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, volume 183 of *Methods in Enzymology*, pages 63–98. Academic Press, 1990.

[PL88] William R. Pearson and David J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the U. S. A.*, 85:2444–2448, 1988.

[SKH93] D. Seto, B.F. Koop, and L. Hood. An experimentally derived data set constructed for testing large-scale dna sequence assembly algorithms. *Genomics*, 15:673–676, 1993.

[WL83] W. Wilbur and D. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences of the U. S. A.*, 80:726–730, 1983.