A Suite of UNIX Filters
For Fragment Assembly

Gene Myers

TR 96-07
A Suite of UNIX Filters
For Fragment Assembly

Gene Myers

TR 96-07

ABSTRACT

A collection of UNIX filters for fragment assembly are presented. The tools are ASCII-based and batch-oriented. The basic repertoire consists of primitives to (1) build a graph of all the overlaps amongst a collection of FASTA-formated fragments, (2) produce any number of alternate assemblies of these fragments from such a graph, (3) display an overlap graph, (4) display an assembly both as a layout of lines representing each fragment, and as a multi-alignment showing the consensus of the fragments, and (5) convert an assembly into a file suitable for input to the interactive editor of XGAP. Tools for additional tasks are planned. This suite is intended to permit an evaluation of our FAKII C-library for fragment assembly and to serve as a set of coding examples of the use of the FAKII library.

April 20, 1996

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

*This work was supported in part under DOE Grant DE-FG03-94ER61911.
A Suite of UNIX Filters for Fragment Assembly

1. Overview

Over the past ten years, we have built a series of libraries of C-coded routines for solving the problem of assembling a collection of shotgun DNA sequence fragments. The most recent version of our Fragment Assembly Kernel, FAKII Version 4.1, consists of 68 routines and is described in a 15 page technical report. These routines provide a flexible framework for solving the problem that can be embedded in any integrated informatics system for DNA sequencing. For example, it is currently being used by other groups to solve dual-end sequencing and transposon-based sequencing problems. However, learning how to use this library takes time and despite our best efforts to carefully document it, conceptual misunderstandings are not uncommon. To alleviate these problems, we have developed a modest set of UNIX filters that employ the FAKII library to perform fragment assembly at the UNIX command level. This suite of tools is simple to use and the source code for them provides an example to other developers of how to use the library.

2. Computing and Storing Overlaps

The first step in assembling a collection of fragments is to determine all pairwise overlaps between them, both in the forward and reverse orientation. The tool create_graph takes a FASTA formatted file of fragments, compares the fragments to each other detecting overlaps, stores the fragments and overlaps between them in an overlap graph where each vertex represents fragment and each edge represents an overlap, and finally writes the overlap graph to the standard output.

```
create_graph {e_limit:%f} {o_thresh:%f} {d_limit:%f} {fragments:FASTA file} > {graph}
```

The performance of create_graph is modulated by three floating point numbers that control which overlaps are detected and/or accepted as follows:

- **e_limit**: The maximum sequencing error rate for which overlaps will be guaranteed to be detected. For example, if this is set to 10%, then the create_graph looks for overlaps with 20% or less differences in the aligned regions. This parameter should never be greater than .2, and we suggest .099 as a standard value (see below).

- **d_limit**: It is further required that the distribution of differences along the alignment of an overlap not be highly skewed but spread across the alignment. The distribution score of an alignment is the minimum over all segments of the alignment of the probability that one would see the observed number of differences in that segment given an underlying error process occurring at rate e_limit. This probability should not be too small, as if it is, it implies there is a segment of the alignment that has an unusually large number of differences in it. Note that this is quite conservative as we are assuming the error process is at the maximum error rate (and not the average error rate). We recommend using a value of .0001 or less.

- **o_thresh**: The overlap score of an overlap is the log of the _a priori_ odds that such an overlap would occur by chance. Pragmatically, this score is the length of the overlap minus a marginally decreasing penalty per difference. A typical value is 10, implying an overlap of at least 10 bases is needed and that for the overlap to occur by chance is a one in a million (≈ 4^10) event.

The e_limit and d_limit parameters control the efficiency with which overlaps are detected. The smaller the error limit or the higher the distribution limit, the less time overlap detection will take. By far the most important of these two efficiency parameters in Version 4.1 is the error_limit. Note that both are not "thresholds", but only "limits": create_graph guarantees to find all overlaps inside the error limit and distribution limit, but may report additional overlaps as well. On the other hand, the overlap threshold is a true threshold: any overlap not scoring above it, i.e., that is not statistically significant enough, will not be entered into the overlap graph.
One should set these three parameters to the most lenient/inclusive values that they think will be ever be needed for proper assembly, moderated by the level of efficiency with which the computation can be done. Philosophically, our view is that overlap detection is a one-time computation in which one determines all the possible ways that the fragments could go together. Later, during assembly (see below), one can select a more stringent subset of the overlaps with which to meld fragments. With regard to efficiency, it should be noted that there are significant changes in performance as e_limit crosses the levels .05 and .10. Thus our recommendation to use .099 as a standard setting.

The FASTA format for sequence files is a standard and very simple and minimalist format. Essentially each sequence entry may be written on several lines. Sequences are separated by one or more “header” lines that begin with ‘>’ but may otherwise contain whatever the user desires. For example,

```plaintext
> GMDA1AA02.seq [Unknown form], 52 bases, 3581 checksum.
CAGGCATAGCTACCTTGCCACCCATTTTTGTTTTTTGTAGAGAT
> GMDA1AA04.seq [Unknown form], 127 bases, 7543 checksum.
TTTTTTGTAAGAGATGGGGTTTTTGTCATGGCCTGGGTCTTGAGAC
TCTGGAGCAGCAGCATTGGCCTCTclockAAATGTGTGGGATT
ATAGGCATAGCGCACCGCAACGct
> GMDA1BA01.seq [Unknown form], 61 bases, 8399 checksum.
AGGAGAATACCTTGAAACCCCGGGAGGTGGAATGTGTAGTGAGCCAAGATCA
CGCAATTGTAC
```
gives three short sequences. For later reference, a name is associated by create_graph with each fragment. The name is the first non-blank sequence of symbols of the first header line preceding the sequence. In the example above, the three fragments would henceforward be referred to as GMDA1AA02.seq, GMDA1AA04.seq, and GMDA1BA01.seq, respectively. If the header line is blank, then create_graph assigns the fragment the name frag(#) where the integer reflects its ordinal position in a list of the fragments in the overlap graph.

Because an input data set of fragments is not always available all at once, the construction of the overlap graph can take place in several stages via the use of the utility add_to_graph.

```
add_to_graph {e_limit:%f} {o_thresh:%f} {d_limit:%f}
{fragments:FASTA file} < {graph} > {graph}
```

This utility reads an overlap graph from the standard input, reads the FASTA file of fragments, compares the FASTA fragments to each other and to every fragment in the overlap graph, and writes an overlap graph representing the augmented result to the standard output.

The utility show_graph reads an overlap graph from the standard input and generates an ASCII readable "print out" of the graph or a selected portion of the graph on the standard output.

```
show_graph [-d] [ {from_frags:reg.exp.} [ {to_frags:reg.exp.} ] ] < {graph}
```

If the two optional regular expression arguments are absent then the entire graph is displayed. If one regular expression is present then all fragments whose name matches the regular expression, and all edges to/from the matched fragments are displayed. If two regular expressions are specified then show_graph displays all edges for which one fragment’s name matches one regular expression and the other fragment’s name matches the other regular expression.

The regular expression mechanism supported is exactly that of the UNIX egrep tool (including ‘?’ for start of string and ‘$’ for end of string) plus three special notations for numbers, identifiers, and number ranges. First the symbol ‘#’ is a short hand for the regular expression “0\[1-9]\[0-9\]*”, that matches any positive integer constant. For example, the pattern “frag#” matches “frag1”, “frag13”, and “frag203” but not “frag033”. To match the latter use the pattern “frag0*#”. Second, the symbol ‘@’ is a short hand for “[@A-Za-z]@[A-Za-z\-0-9]*”, i.e., any C-identifier. Finally, there is support for a number class, denoted “1{integer_1} . integer_2}””. The first integer must not be greater than the second. The number classes matches any sequence of digits denoting a number in the range of the class. For example “1{3-56}” matches “3”, “4”, ..., “9”, “10”, “11”, ..., and “56”. The integers in the number class may be padded on the left with zeros in which case the padding is considered to be part of the matching string. For example, “1{003,056}” matches “003”, “004”, ..., “009”, “010”, “011”, ..., and “056”.

- 2 -
If the -d option of show_graph is set then the report includes the sequences for each fragment and a display of the alignment for each overlap between a pair of fragments. Otherwise the report simply lists the fragment names and the overlaps between them. A portion of a sample output appears as follows:

*** BEGIN GRAPH DISPLAY *****

Range of parameter values edges were created with:
   e_limit: 0.099
   o_thresh: 10
   d_limit: 1e-05

Fragment = frag1
Length = 346
Sequence:
   caagacctgtaatcttacactgttctcatgctatccaccaaccta
tttcagaggaacacgtaagggagtagtagtaatagcgccttctcactca
gcatagcaacactgactgtcgcagcatctgggtgatgggtttttgatc
tagacacccccccgtaatgttaaatagatgcacacaggagaccaggtcaggg
   atataaaacctctacgaacgggagaccgtacgcatctagatgttgaata
   aataaacctgtacgctagttcgcagttgagagacccgtatcaggg
tgcaccaacataagacccaggtatatcagctcaaacgctagcgtcaggt
Inedges:
   <+--< frag17 (o_score = 222.4, d_score = 0.5657)
   <===> frag297 (o_score = 57.0, d_score = 0.5657)
   <===> frag127 (o_score = 136.7, d_score = 0.5067)
Outedges:
   >====< frag28 (o_score = 23.2, d_score = 0.7190)
   >-----< frag141 (o_score = 195.2, d_score = 0.6474)
   >-----< frag140 (o_score = 129.3, d_score = 0.8338)

*** EDGE LISTINGS:

Edge: frag1 <==== frag297 (o_score = 57.0, d_score = 0.5657)
    
    78
    ---------> frag1
frag297 <--------
    
    76

   frag1>: caagacctgtaatcttacactgttctcatgctatccaccaaccta
          --------------------------------------------
   frag297<: caagacctgtaatcttacactgttctcatgctatccaccaaccta

   frag1>: ttgctcagaggaacacgtaagggagtagta
          --------------------------
   frag297<: ttgctcagaggaacacgtaagggagtagta


***** END GRAPH DISPLAY ***

The quantities o_score and d_score reported for each edge are its overlap score and distributional score. Edges/overlaps in the overlap graph are of 8 different types depending on the “geometry” of the overlap. The table below gives the meaning of each edge denotation one will find in a printout.
One need not memorize these 8 types as a graphic of each edge relationship is given in a -d listing. For example, in the output above, the edge fraq1 <=== fraq297 overlaps the first 78 symbols of fraq1 with the first 76 of fraq297 when it is oriented opposite to that of fraq1.

3. Computing and Displaying Assemblies

Once an overlap graph has been constructed, one may then proceed to assemble the fragments. The tool assemble accomplishes this task by reading an overlap graph from the standard input and writing an assembly object to the standard output.

assemble [-{ith:%d}] [-{c{constraints:file}}] 
{e_rate:%f} {o_thresh:%f} {d_thresh:%f} < {graph} > {assembly}

An assembly object records the relative positions of all fragments in the assembly, the multi-alignment of the fragments, possibly in a number of disjoint contigs, and the consensus sequence for the multi-alignment. The assembly takes place over a subset of the edges in the overlap graph determined by three floating point parameters as follows:

e_rate The distribution score of each edge in the overlap graph will be computed assuming an error process at the specified rate. Edges will then be eliminated if their distribution score/probability is below d_thresh described next.

d_thresh Specifies the minimum error distribution score for edges to be considered in assemblies. Any edge in the overlap graph whose distribution score with respect to error rate e_rate is less than d_thresh is eliminated from consideration as regards melding fragments. Setting this parameter to 1.0 eliminates all edges, and setting it to 0.0 eliminates none.

o_thresh Specifies the minimum overlap score for edges to be considered in assemblies. Setting this parameter to 0. guarantees that no edges are eliminated on this basis.

For example, if a graph was created with the command:

create_graph .099 10. .0001 input >graph

then the command:

assemble .05 20. .0001 <graph >assembly

assembles over a subset of the graph whose overlap scores are greater than 20 and whose distributional scores at rate 5% are greater than .0001. Further note that the command:

assemble 0. 0. 0. <graph >assembly

is guaranteed to assemble over all edges in the overlap graph.

There are two optional parameters to assemble. The FAK kernels have always been able to generate a series of alternate solutions to the problem in decreasing order of ‘‘goodness’’. A parameter of the form, e.g., -3, would produce the third best assembly. The assembler can also produce solutions consonant with a collection of constraints with the -c{constraints} option, where constraints is presumed to be a file containing the results of a fac_write primitive. As yet our UNIX suite does not contain any tools for producing such a file, but a user may use the FAKII kernel to produce one if desired.

There are two routines for producing displays or printouts of assemblies. The first is show_layout which produces a ‘‘stick diagram’’ of an assembly in which the arrangement of fragments in each contig of an assem-
bly is shown by depicting each fragment as a line with an arrowhead at one end or the other to indicate its orientation.

```
show_layout [-c{contig:%d}] [-w{columns:%d}] [-z{scale:%d}] [-n] < {assembly}
```

The assembly to be displayed is read from the standard input. The number of columns used for the display can be controlled by the optional \(-w\) option which defaults to 50. The “zoom” factor of the display can be controlled by the optional \(-z\) option which specifies the number of bases to be represented by a single column (default is 20). The \(-n\) option, if set, turns off the display of fragment names. Finally, only a single contig of an assembly may be displayed by giving the contig number following the \(-c\) option. An example of a part of the output obtained by the command `show_layout -w70 -z30 <assembly` is as follows:

```
*** BEGIN LAYOUT DISPLAY ***

Range of parameter values edges were created with:

- e_limit: 0.099
- o_thresh: 10
- d_limit: 1e-06

This is the best assembly.

Parameter values are:

- e_rate: 0.04
- o_thresh: 15
- d_rate: 0.0001

*** CONTIG 1 (Score = 35226):

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```
A dot at the left boundary of the layout indicates that the fragment adjacent to the dot is continued from the preceding “frame”. Similarly, a dot at the right boundary indicates that the adjacent fragment continues on the succeeding frame. Given an ASCII display, the best we could do for indicating the names of each fragment was to give for each row of a frame, the list of fragments on the row of that frame in order of their first column position. This permits easy identification in conjunction with the scale bar at the top of each frame.

The second assembly display routine, show_multi, prints a multi-alignment of each contig of an assembly, along with the consensus sequence by default.

show_multi [-c{contig:%d}] [-w{columns:%d}] [-m] < {assembly}

The -c and -w options are as for show_layout. In addition the -m option, if set, suppresses the printout of the consensus sequence. A sample output of show_multi is as follows.

*** BEGIN MULTI-ALIGNMENT DISPLAY ***

*** CONTIG 1 (Score = 15670.7):

*** END MULTI-ALIGNMENT DISPLAY ***
4. Summary

The basic suite described above consists of six commands which are formally summarized as follows.

1. create_graph \{e\_limit:%f\} \{o\_thresh:%f\} \{d\_limit:%f\}
   \{fragments:FASTA file\} > \{graph\}

2. add_to_graph \{e\_limit:%f\} \{o\_thresh:%f\} \{d\_limit:%f\}
   \{fragments:FASTA file\} < \{graph\} > \{graph\}

3. show_graph [-d] \{from\_frags:reg.exp.\} \{to\_frags:reg.exp.\} \{graph\}

4. assemble [-\{\text{ith}:%d\}] [-c\{constraints:file\}]
   \{e\_rate:%f\} \{o\_thresh:%f\} \{d\_thresh:%f\} < \{graph\} > \{assembly\}

5. show_layout [-c\{contig:%d\}] [-w\{columns:%d\}] [-z\{scale:%d\}] [-n] < \{assembly\}

6. show_multi [-c\{contig:%d\}] [-w\{columns:%d\}] [-m] < \{assembly\}

The conventions for parameters in these descriptions are as follows. Any parameter in square braces is optional. Any meta-item is enclosed in curly braces and within the braces is a one word description of the item followed by a colon and a syntactic specification of the item, e.g. %d denotes an integer. Any redirected item is either from/to another stage of a pipeline or is a file.

5. References

[LJAM96] Larson, S., Jain, M., Anson, E. and Myers, E. ‘An Interface for a Fragment Assembly Kernel.’
http://www.cs.arizona.edu/research/reports.html.