Interface for the FAKII Fragment Assembly Kernel

Susan Miller
Mudita Jain
Eric Anson
Gene Myers

TR 99-01

Interface for the FAKII
Fragment Assembly Kernel*

Susan Miller
Mudita Jain
Eric Anson
Gene Myers

TR 99-01

ABSTRACT

This document describes the C programming language interface to our FAKII Fragment Assembly Kernel library. Inputs to the Fragment Assembly Kernel are (1) DNA fragment sequences from potentially inaccurate sequencing experiments, and (2) optional constraints on fragment assembly such as known fragment overlaps or relative fragment orientation. Fragment sequence version control is supported. The Fragment Assembly Kernel produces the most probable reconstructions of the original DNA sequence from the fragments, subject to any specified constraints. Each fragment assembly includes multiple sequence alignment and consensus sequences. Multiple sequence alignment editing capabilities are provided to allow manual correction of sequence errors.

February 3, 1999

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported in part by DOE Grant DE-FG03-94ER61911.

Interface for the FAKII Fragment Assembly Kernel

1. Overview

At a conceptual level, the problem of assembling DNA sequence fragments naturally divides into three phases. In the *overlap phase* each fragment is compared against every other fragment to see if they share a common subsequence, implying that they were potentially sampled from overlapping stretches of the original strand. Each pair of fragments is compared in two ways: with both fragments in the same relative orientation, and with one of the fragments having been reverse-complemented. The result of this first phase may be thought of as an *overlap graph* in which each fragment is modeled as a vertex and each statistically significant overlap between two sequences is modeled as a directed edge between the vertices representing them.

The second, *layout phase* takes the overlap graph as input and generates a series of alternate assemblies or layouts of the fragments based on the pairwise overlaps therein. A layout specifies the relative locations and orientations of the fragments with respect to each other and is typically visualized as an arrangement of overlapping directed lines, one for each fragment. The general criterion for the layout phase is to produce plausible assemblies of maximun likelihood, but with the advent of mixed-mode sequencing strategies, may also be required to meet an additional *set of constraints*. We advocate that the layout algorithm must be generative, i.e. produce a sequence of layouts in decreasing order of "quality". For example, it is important to know if there is more than one way to put the pieces together, especially if different solutions appear equally plausible. In such a case, one would return to the lab and obtain additional information to remove the ambiguity.

The final, *multi-alignment* phase uses more information than just the pairwise alignments in the layout. The sequences of all the fragments in a layout are simultaneously aligned, giving a final consensus sequence as the desired reconstruction of the original strand. We think of these final multi-alignments as being a resulting *assembly*.

The Fragment Assembly Kernel (FAKII) facilitates the creation of three types of objects: *overlap graphs*, *constraint sets*, and *assemblies*. Overlap graphs record versions of fragment sequences and the overlaps between them. Constraint sets store information about fragment relationships, such as fragments that are known to overlap or to not overlap, or fragments that are in the same orientation or are reverse complemented with respect to each other. For a given overlap graph and an associated constraint set, a series of fragment assemblies can be generated. In generating an assembly, fragments are assembled into *contigs*, or groups of overlapping fragments, and a multi-alignment is computed for each contig. Each successive assembly is built up from a different "seed" edge from the overlap graph, to increase the likelihood that the resulting assembly is different from any preceding assemblies.

Functions are provided to create and destroy overlap graphs, constraint sets, and assemblies, and to read and write them to and from disk files. Each object created by a Fragment Assembly Kernel function is persistent until it is destroyed by another FAKII function. FAKII routines allow access to information about fragments in a contig, such as fragment position, orientation, and type of overlap. The Fragment Assembly Kernel provides functions for editing the multiple sequence alignments so that errors in fragment sequences may be corrected manually.

The FAKII 4.2 kernel features a new bit-vector based comparison algorithm devised by Gene Myers for much greater efficiency in the overlap computation phase. Comparison stringency has also been reduced to make overlap detection more sensitive.

2. Initialization, Error Handling, and Shutdown of the Kernel

```
int   fa_startup (int trapflag, char *path);
char *fa_error_msg ();
void  fa_shutdown();
```

Function `fa_startup` initializes the fragment assembly system and must be the first routine called. If `trapflag` is non-zero, `fa_startup` uses the C Library *setjmp/longjmp* mechanism to allow control to return to the point of the call to `fa_startup` in the event of an error. The first call to `fa_startup` returns zero but has the important side effect of establishing itself as the return point for error exceptions. Thereafter, whenever an error is detected, control is transferred to the call as if it had just returned, but this time with a non-zero value indicating the type of error. In this way error handling is left to the discretion of the user of the kernel. The user routine that calls `fa_startup` must not return before invoking other Fragment Assembly Kernel procedures, otherwise the system may be returning control to a non-existent environment. See the example below and refer to the C library function *setjmp* for a description of this mechanism. If `fa_startup` is called with `trapflag` set to zero, then on the detection of an error, an error message is output on stderr and execution terminates. The `path` string passed to `fa_startup` must be the pathname of the directory in which the FAKII score table file(s) (fa*.i) reside. If `path` is a NULL pointer or an empty string (""), FAKII will expect the score table file(s) to be in the current directory.

Function `fa_error_msg` returns a pointer to a string containing the error message for the most recently detected error. The following code fragment is an illustration of the use of the FAKII error routines:

```
int rc;

if (rc = fa_startup(1, ""))
{
   fprintf(stderr, "%s", fa_error_msg());
   ...error handling based on the value of rc...
}
else
{
   ...calls to other Fragment Assembly Kernel routines...
   fa_shutdown();
}
```

The `fa_shutdown` procedure frees working memory used by the kernel for overlap computations, minimizes the amount of memory allocated for error checking based on the number of extant graph, assembly and constraint objects, and removes the file created for pointer checking. This routine may be called at any time to free memory, and any subsequent calls to FAKII routines needing the freed structures will result in their being rebuilt on a demand basis.

3. Constructing Overlap Graphs

An overlap graph is constructed by using FAKII primitives to perform a series of additions and deletions of fragments and edges between the fragments. Each edge in the overlap graph represents one of two types of overlap. A *containment* overlap between two fragments occurs when one fragment sequence is completely contained within the other fragment sequence. An overlap between a suffix of one fragment and a prefix of another is called a *dovetail*.

All possible alignments are represented with one of the following edge types:

```
A --------------
      -----      B     (A contains B)


A -------------- _
      -----      B     (A contains B reverse complement)


B --------------
      -----      A     (B contains A)
```

```
_
B --------------
      -----      A     (B reverse complement contains A)


A ---------------
      ---------------- B    (A dovetails to B)


A ---------------      _
      ---------------- B    (A dovetails to B reverse complement)


B ---------------
      ---------------- A    (B dovetails to A)

_
B ---------------
      ---------------- A    (B reverse complement dovetails to A)
```

Note that, for example, (A reverse complement contains B) is not in the list, but it can be represented by (A contains B reverse complement), which is an encoding of the same alignment with the fragments in the opposite orientation. To simplify the encoding, FAKII edge representations always refer to the A fragment in its forward orientation, and the B fragment may or may not be reverse complemented. This is reflected in the edge types listed above.

FAKII uses an edge edit script to represent each alignment between two fragments. The edit script is an array of integers that encodes the alignment. The first integer is FA_SAME if the B fragment is in its forward orientation, and FA_COMP if the B fragment is reverse complemented. The second and third integers represent the left and right ends of the overlap as indicies of bases in the fragments. Positive indices are in the A fragment, and negative indices correspond to bases in the B fragment. Positions are numbered from left to right, 1 to the length of the fragment, and is a fragment is reverse complemented, the numbering is done afterwards. Inserts and deletes do not count as positions in fragments. If both indices are in the same fragment (i.e. have the same sign), the edge is a containment, otherwise it is a dovetail. The remaining integers in the script form a list of insertions and deletions, with a zero at the end of the list. Again, positive integers refer to positions in the A fragment and negative integers to positions in the (possibly reverse complemented) B fragment. An insert or delete position is the index of the base that comes after the insert or delete.

As an illustration, the edit scripts for a few example edges are:

```
       i   j           (i = 5, j = 8)
A  gtgcta-gtgatgc                      edit script: (FA_SAME,5,8,7,0)
       taggt      B     (i * j > 0 => containment)


      -i  -j           (i = -5, j = -8)
B  ccgcgat-aggcac                      edit script: (FA_SAME,-5,-8,2,-8,0)
       g-tta      A     (i * j > 0 => containment)
```

```
  _     -i                    (i = -5, j = 10)
  B  atatgtca-tagtgc                     edit script: (FA_COMP,-5,10,2,2,-9,0)
         g--attagtgccaat  A    (i * j < 0 =>  dovetail)
                   j
```

In determining significant overlaps, a user-specified *error rate* is taken into account. The error rate is multiplied by the sum of the lengths of the two fragments being compared, giving the maximum number of differences allowed in an overlap that is to be represented in the graph. That is, if the error rate is 5% then a sequence of length 500 could have 25 errors in it, which implies that when comparing two such sequences up to 50 differences must be permitted:

```
                  |------  up to 25 errors in fragA  ------|
          1                                               500
fragA     ==========xx====x===xx=x==x=x===xx===x== ... ===
fragB             ======xx==xx==xx==x==xxx==x=xx==x ... ==========
                  1                                               500
                  |------  up to 25 errors in fragB  ------|
```

We have always advocated using full-sensitivity sequence comparisons for finding approximate overlaps, as opposed to heuristics which occasionally miss significant overlaps. Sequencing errors must be accommodated and while one may not wish to use the more error laden data toward the end of a gel run for the purposes of multi-alignment and consensus, its use for detecting overlaps can significantly improve closure probabilities for pure shotgun projects. Using higher error rates for overlap comparisons allows for less trimming of the raw data. Thus we argue that the further ability of our approach to correctly handle *large* error rates is an asset.

In order to support version control for fragments, the set of fragments in a graph are partitioned into *classes*, each class representing the versions of a given fragment. Only one fragment in a class can be *active* and it is this fragment that is used in overlap comparisons and in assemblies. Before generating assemblies over a graph, the active fragment of a class can be changed, fragments can be added and deleted from a class, and classes can be added and deleted from the graph. If a user does not intend to support version control, then they can simply place one fragment in each class.

The following primitives can be used to construct overlap graphs.

```
FA_GRAPH *fa_create_graph ();
void      fa_destroy_graph (FA_GRAPH *graph);
```

Function `fa_create_graph` returns a pointer to an empty overlap graph. The routine `fa_destroy_graph` frees the memory consumed by an overlap graph.

```
FA_NAME fa_add_class (FA_GRAPH *graph, char *sequence, int ext_id,
                      int cmp_type, double error_limit,
                      double overlap_threshold, double distrib_limit);
void    fa_del_class (FA_GRAPH *graph, FA_NAME frag);

FA_NAME fa_add_frag (FA_GRAPH *graph, FA_NAME frag, char *sequence, int ext_id,
                     int cmp_type, double error_limit,
                     double overlap_threshold, double distrib_limit,
                  int *trace_edit_script);
void    fa_del_frag (FA_GRAPH *graph, FA_NAME frag);

void fa_active (FA_GRAPH *graph, FA_NAME frag);

void fa_list_class (FA_GRAPH *graph, FA_NAME frag, void (*handler)());
     void handler (FA_NAME fname);

void fa_list_active (FA_GRAPH *graph, void (*handler)());
     void handler (FA_NAME fname);
```

Function `fa_add_class` establishes a new fragment class whose sole and active member is the supplied sequence with associated user-supplied id, `ext_id`. The fragment is assigned a name of type `FA_NAME` that is the return value of the call and that must subsequently be used to refer to the fragment in calls to FAKII routines. The remaining parameters control the addition of edges between this new fragment and the active members of other classes in the graph. If `cmp_type` is `FA_COMPARE_NONE` then no edges are added and the remaining parameters need not be specified. If `cmp_type` is `FA_COMPARE_ALL` then the new fragment is compared against all active fragments. The overlap computation is guaranteed to produce all overlaps within the specified `error_limit` and `distrib_limit`, and those with a score greater than or equal to the `overlap_threshold` are added to the overlap graph. The overlap computation may produce some overlaps that are outside of the `error_limit` or `distrib_limit` range; if desired these may be screened out for the purpose of assembly with the parameters passed to `fa_init_assemble`. Note that the `error_limit`, `distrib_limit` and `overlap_threshold` values apply to the overlap comparisons done by `fa_add_class`, rather than being associated with the fragment being added to the overlap graph. When a fragment is added to an overlap graph, these values are used to determine the edges that are added to the graph. A detailed description of overlap and error-distribution scores is given below, in the discussion of the `fa_compare` function.

The `fa_del_class` procedure removes from the overlap graph all fragments in a class and the edges incident to these fragments.

Function `fa_add_frag` is identical to `fa_add_class` except that (1) the supplied fragment sequence is added to the pre-existing class containing fragment `frag`, (2) the parameter `cmp_type` may be specified as `FA_COMPARE_CLASS`, and (3) a trace edit script may be specified. To illustrate the use of the `cmp_type` parameter, let A be the active fragment in the class. If `cmp_type` is `FA_COMPARE_CLASS`, then the new fragment is compared against the active fragments of the classes containing fragments to which A has an edge. Just before `fa_add_frag` returns, the new fragment is made the active fragment of the class.

The trace edit script passed to `fa_add_frag` by the parameter `trace_edit_script` specifies an alignment between the version of the fragment being added and the trace from which the fragment originally came. This information is used by routines `fa_align_trace` and `fa_align_frag` (described later) which given a basepair in an edited fragment return the corresponding location in the trace (or vice versa). If one is not interested in this information the NULL pointer may be passed in for `trace_edit_script`.

When `fa_add_class` is called a trace edit script is automatically created which specifies an alignment between the fragment and its trace which contains no gaps. When a fragment is used in an assembly a copy of its trace edit script is made. This copy is automatically updated as the fragment is edited within the assembly. If one then wishes to use `fa_add_frag` to save this edited fragment as a new version, one would call the routine `fa_get_trace` (described later) to obtain the trace edit script associated with this version of the fragment. The value returned by this routine would then be passed into `fa_add_frag`. Note that with this envisioned usage one need not be concerned with knowing the format of the trace edit script. To be complete, however, we specify the format below.

The format of the trace edit script is the same as the edge edit script with the trace corresponding to fragment B and the fragment corresponding to fragment A. There are, however, the following differences. The fragment and its trace are always assumed to have the same orientation. The first element of the the trace edit script is the size of the array in which it is stored. The second element is the size of the edit script itself. (This is generally smaller than the size of the array to allow room for growth.) The edit script (i.e. the specfication of the dashes in the alignment) follows.

The procedure `fa_del_frag` removes the specified fragment `frag` from its class and removes any edges incident to `frag` from the overlap graph. If `frag` was the active fragment of the class, then another fragment in the class is randomly selected to be active.

The routine `fa_active` makes the specified fragment the active fragment of the class containing it. Recall that only one version of a fragment can be active at any given time.

The procedure `fa_list_class` may be used to list all of the fragments in the class containing fragment `frag`. The user writes a `handler` routine that is passed to `fa_list_class`. The handler is called once for each fragment in the class, with the `FA_NAME` of the fragment passed in as a parameter.

The `fa_list_active` procedure may be used to list the active fragments in the overlap graph pointed to by `graph`. The user writes a `handler` routine that is passed to `fa_list_active`. The handler is called once for each fragment class in the graph, with the `FA_NAME` of the active fragment of the class passed as a parameter.

In Version 4.0 or later of the FAKII kernel, searches against a set of fragments as implied by the use of `FA_COMPARE_ALL` in the calls to `fa_add_class` and/or `fa_add_frag`, can be significantly accelerated by opting to use a large index structure in conjunction with some new code. The single instance of this index is allocated by the primitive `fa_create_the_index` and destroyed with the primitive `fa_destroy_the_index` below. The former must be called after `fa_startup` is called, and the latter before `fa_shutdown` is called. While the index is in existence, one can associate it with a given graph, `graph`, by invoking `fa_apply_index` on it. While the association between `graph` and the index is in effect, all `FA_COMPARE_ALL` searches over the current set of active fragments in the graph will be accelerated. One is free to add and delete fragments and classes, or to change the active fragment in a class, during the time of association: the changes are automatically reflected in the index.

```
void fa_create_the_index ();
void fa_destroy_the_index();
void fa_apply_index (FA_GRAPH *graph);
```

The efficiency of the overlap computation is primarily related to the error rate specified when fragments are added to the graph. Lower error rates will allow a more efficient overlap computation. The most efficient overlap computation is possible when the error rate is low, say less than 5%. A somewhat lower efficiency results at or above each of the 5%, and 10% error rate levels. Refer to the section on Building and Using the Fragment Assembly Kernel for more details regarding efficiency and index memory requirements.

To examine the results of an overlap comparison between two fragments, `fa_compare` may be used.

```
int *fa_compare (FA_GRAPH *graph, FA_NAME frag1, FA_NAME frag2,
                 double error_limit, double *score, double *distrib_limit,
                 int orient, int coord1, int coord2);
```

Function `fa_compare` returns a pointer to a list of integers representing an optimal overlap alignment between `frag1` and `frag2`. The overlap is computed using the specified error rate and any non-zero values in the `orient` and `coord1`, `coord2` parameters. Passing non-zero values for `orient`, `coord1` and `coord2` specifies a more restricted set of comparisons, as explained below. In `fa_compare` the highest scoring overlap (consistent with `orient`, `coord1`, and `coord2` values, if any) is determined, and the overlap score and error-distribution score are returned via the pointers `score` and `distrib_limit`, respectively. If no such overlap is found, `fa_compare` returns NULL.

The overlap score of an alignment roughly reflects the length of the overlap with a deduction for mismatches in the alignment. The score is computed as $-\log_4$ of the probability that such an alignment would occur at random. This is a prior odds ratio, i.e. it does not take into account the total number of comparisons made in building a particular overlap graph. By taking the $-\log_4$ of the probability, scores are scaled so that a perfect alignment of length L, has score L, and an alignment with D errors in it scores approximately $L - D \log_4 L/(D+1)$ when D/L is sufficiently small. That is, the score is the length of the overlap less a factor multiplied by the number of differences, where the factor becomes smaller as the number of difference becomes larger. We find that choosing a cutoff score of around 10 (1 in a million) is generally satisfactory.

The error-distribution score of an alignment provides another useful and orthogonal measure of the quality of an overlap. It is based on the Erlang approximation of the probability of seeing the number of differences in the alignment given that errors are distributed exponentially with arrival rate `1/error_limit`. The need for this additional measure is motivated by the following example. Function `fa_compare` will find all overlaps involving up to `error_limit*(length(frag1)+length(frag2))` differences. So one might find an overlap of 200 symbols between two fragments of length 500 with 100 differences when the error rate is set at 10%. The overlap score in this case is still well over 10 as such an alignment is very rare. But if the 10% errors were distributed exponentially we would on average see only 40 errors in an overlap of 200 symbols, and would see 100 errors in this overlap only 1 in a 1000 times. Thus the Erlang-based error-distribution score reveals such an alignment to be suspect. Another common phenomenon is for a significant overlap (with respect to overlap score) to occur in the case of a chimeric fragment or two fragments that both contain part of a repeated sequence in the original DNA strand. In these cases one has a very good alignment, distributionally speaking, for a prefix or suffix of the overlap followed by a very poor alignment thereafter. We capture this by computing the distributional score of every suffix and prefix of an alignment and taking the minimum score over all. This number is the distributional score returned by `fa_compare`. When passed as a parameter to the `fa_add` routines, a typical distrib_limit threshold is about `.001`. An error-distribution limit of `.01` tends to eliminate some good alignments, and `.0001` tends to retain potentially bad ones. Note that using `0.0` guarantees that no edges are eliminated on this basis, and using `1.0` is non-sensical as it will guarantee that all overlaps are rejected.

If the fragments are to be compared in both the same and reverse complement orientations, the `orient` parameter should have a zero value. If one wishes to specify the relative orientation of the fragments considered, the parameter `orient` is used as follows: Setting `orient` to `FA_SAME` specifies that `frag1` and `frag2` are to be compared in the same orientation. If `orient` is `FA_COMP`, then `frag1` is compared to the reverse complement of `frag2`. If the orientation constraint is given and the last two arguments have a zero value then the best overlap between the fragments in the given orientation is returned.

Finally, if the best alignment subject to a given orientation and overlap interval is desired, then one may further specify the interval with the parameters `coord1` and `coord2`. These parameters specify the beginning and ending positions of the overlap interval. Positive values represent positions in `frag1`, and negative values indicate positions in `frag2`. If the product of the values for `coord1` and `coord2` is positive, a *containment* overlap is indicated; if this product is negative, it represents a *dovetail* overlap. For example, if `frag1` is 110 characters long and `frag2` is 60 characters long, then the (`orient`, `coord1`, `coord2`) triples at left give rise to the overlaps at right:

```
(FA_SAME,100,-10) <==> frag1[100..110] dovetails frag2[1..10]
(FA_SAME, 20, 85) <==> frag1[ 20.. 85] contains  frag2
(FA_SAME,-45, 20) <==> frag2[ 45.. 60] dovetails frag1[1..20]
(FA_SAME, -1,-60) <==> frag2 contains frag1 (with lots of errors)
(FA_COMP,100,-10) <==> frag1[100..110] dovetails FRAG2[1..10]
(FA_COMP, 20, 85) <==> frag1[ 20.. 85] contains  FRAG2
```

where `FRAG2` is the reverse complement sequence of `frag2`

Notice that `FRAG2[1..10]` is the reverse complement of `frag2[51..60]`. Also notice that the overlaps may contain errors. For example, the first triple above implies that the last 11 characters of `frag1` overlap with the first 10 characters of `frag2`, so there must also be an insert in `frag2` in order for it to align with `frag1`.

The list of integers returned by `fa_compare` represents the alignment as follows: The first three integers indicate the orientation and overlap interval, exactly as do the parameters to `fa_compare`. That is, the first integer in the list is `FA_SAME` or `FA_COMP`, indicating whether or not `frag2` is reverse-complemented. The second and third integers indicate the beginning and ending positions of the overlap interval as described above. The remaining integers in the list indicate the positions at which to insert dashes into the two sequences so as to produce the encoded alignment. Specifically, a positive integer, k, indicates that a dash should be inserted before the k'th symbol of `frag1`, and a negative integer, –k, indicates that a dash should be inserted before the k'th symbol of `frag2`. The list is terminated with a 0. For example, if `frag1` = `'acggtacgttacgatacg'` and `frag2` = `'gtaaacttaagaacgtaa'`, then the alignment:

```
acggt--acgttacgatacg
   gtaaac-ttaaga-acgtaa
```

is specified by the list <`FA_SAME`,4,–15,6,6,–7,–13,0>.

The following FAKII routines may be used to add and delete edges from an overlap graph manually, and to inspect the edges in a graph:

```
void fa_add_edge (FA_GRAPH *graph, FA_NAME frag1, FA_NAME frag2,
                  int *alignment, double o_score, double ed_score);
void fa_del_edge (FA_GRAPH *graph, FA_NAME frag1, FA_NAME frag2,
                  int orient, int coord1, int coord2);


void fa_list_edges (FA_GRAPH *graph, FA_NAME frag, void (*handler)());
     void handler (FA_NAME fname1, FA_NAME fname2,
                   int *alignment, double o_score, double ed_score);
```

The procedure `fa_add_edge` adds an edge from `frag1` to `frag2`, using the specified alignment and overlap and error-distribution scores, to an overlap graph. The alignment is represented by a list of integers such as those returned by `fa_compare`. The edge is assigned the designated scores for the purposes of computing best layouts. These may be the scores returned by `fa_compare` or whatever the user desires (e.g. the length of the overlap for `o_score`).

The `fa_del_edge` routine removes from an overlap graph all edges between `frag1` and `frag2` with the specified orientation (`FA_SAME` or `FA_COMP`) and overlap interval (`coord1` and `coord2` as described for `fa_compare`). That is, the edge is identified by the first three integers in the list of integers encoding it. If there is more than one edge between `frag1` and `frag2` satisfying the description (but possibly differing in the exact alignment between the overlapped intervals), they all are removed.

The routine `fa_list_edges` can be used to obtain information about all edges incident to the specified fragment in an overlap graph. The user writes a `handler` routine that is passed to `fa_list_edges`. The `handler` routine is called once for each edge incident to the specified fragment, and is passed the `FA_NAME`s of the overlapping fragments represented by the edge, a pointer to the integer list encoding of the alignment (as described for `fa_compare`), and the overlap and error-distribution scores of the edge. The `handler` routine can then use this information as desired by the user. The value of either `fname1` or `fname2` passed to the handler is the value `frag` passed to `fa_list_edges`. The handler must expect a score of type double, to allow us to accomodate pre-ANSI C compilers.

```
char *fa_sequence (FA_GRAPH *graph, FA_NAME name);
int   fa_length (FA_GRAPH *graph, FA_NAME name);
int   fa_ext_id (FA_GRAPH *graph, FA_NAME name);
```

Function `fa_sequence` returns a pointer to the character string for the sequence in the overlap graph with the associated `name`, or the null pointer, if no such sequence exists.

Function `fa_length` returns the length of the named sequence, or 0 if there is no such sequence.

Function `fa_ext_id` returns the integer id supplied by the developer when the specified sequence was inserted into the overlap graph. If there is no such sequence, `fa_ext_id` returns 0.

```
void       fa_write_graph (FA_GRAPH *graph, FILE *stream);
FA_GRAPH  *fa_read_graph (FILE *stream);
void       fa_ascii_write_graph (FA_GRAPH *graph, FILE *stream);
FA_GRAPH  *fa_ascii_read_graph (FILE *stream);
```

Procedure `fa_write_graph` stores an overlap graph in a file. The specified file must be opened for writing when `fa_write_graph` is called. Function `fa_read_graph` reads a previously stored overlap graph from a file that has been opened for reading. The `fa_read_graph` and `fa_write_graph` routines make use of the C library buffered I/O functions fread and fwrite. If calls to FAKII read/write routines are intermixed with input or output of other data, these other reads and writes must also use the buffered I/O functions. That is, calls to the system read/write functions cannot be intermixed with calls to the C library fread/fwrite routines.

Procedure `fa_ascii_write_graph` stores an ASCII representation of an overlap graph in a file. The specified file must be opened for writing when `fa_ascii_write_graph` is called. The first line of the ASCII graph file has the format "G(verification code): internal graph structure values". The first two values are the number of classes and the number of fragments in the graph. Next the ASCII representation of each fragment is given as "F: internal fragment structure values" and "S: fragment sequence, 60 characters per line". Class active fragment values and nameindex values follow the fragment data. Finally, for each edge in the graph there is data in the format "E: internal edge structure values including edge overlap score and error-distribution score" and "D: orientation overlap-coordinates" followed by any remaining edit script values (insert positions), 10 per line". Refer to the description of alignment representation in `fa_compare`. Function `fa_ascii_read_graph` reads a previously stored ASCII representation of an overlap graph from a file that has been opened for reading and returns a pointer to the graph.

4. Fragment Assembly Constraints

An additional capability provided by our Fragment Assembly Kernel allows the user to provide more information to the kernel regarding fragment assembly. *Constraints* can be used to specify that given fragments or edges in an overlap graph are used in an assembly in a particular way, or are not included in an assembly.

The inclusion of fragment assembly constraints in the kernel was motivated by the use of mixed-mode sequencing strategies. Previously we had developed an approach for the layout phase that was suitable for pure shotgun sequencing projects [Kec91]. This approach is based on operations research techniques for finding a maximum weight *Hamiltonian path* through the overlap graph of the first phase. Since that time, it has become clear that large sequencing projects will not and cannot employ a pure shotgun strategy. Most experimentalists advocate shotgunning to the point of marginal return and then resorting to primer-based or directed methods for achieving completion or closure. Others advocate approaches involving sequencing only those fragments that do not hybridize (overlap) with other sequenced fragments, or sequencing both ends of an insert, all in an attempt to improve on the coverage of pure shotgunning. The impact of these mixed-mode sequencing strategies is that one must now produce the most compact layout subject to a collection of constraints modeling the additional information provided by the enhanced strategy.

Given that the simple and heuristic greedy algorithm [Sta82,PSU84,Hua92] for producing layouts tends to work well in most cases, and in light of the additional complexity of constraints, we have chosen in our new kernel to utilize a greedy algorithm that will produce solutions that meet the given constraints. Like the basic greedy algorithm, fragments are progressively melded together, where melds are chosen in order of the "degree" of overlap between fragments. But in addition the algorithm rejects a potential meld if it violates a constraint.

The following functions can be used to build *constraint sets*:

```
FA_CSET *fac_all_frags();

FA_CSET *fac_frag_in (FA_NAME frag);
FA_CSET *fac_frag_out(FA_NAME frag);

FA_CSET *fac_edge_in (FA_NAME frag1, FA_NAME frag2,
                      int orient, int coord1, int coord2);
FA_CSET *fac_edge_out(FA_NAME frag1, FA_NAME frag2,
                      int orient, int coord1, int coord2);

FA_CSET *fac_orient_same(FA_NAME frag1, FA_NAME frag2);
FA_CSET *fac_orient_opp (FA_NAME frag1, FA_NAME frag2);

FA_CSET *fac_distance(FA_NAME frag1, FA_NAME frag2, int anchor1,
                      int anchor2, int mingap, int maxgap);
```

Each of the above constraint functions returns a reference to a constraint set containing a single constraint. Constraint sets consisting of more than one constraint can be built using the function `fac_union` which is described below. Fragment constraints have the highest priority, followed by edge constraints, orientation constraints, and finally distance constraints. Therefore, if a fragment is constrained to be out of an assembly, and an edge involving that fragment is constrained to be in the assembly, the edge constraint is considered to be inconsistent with the higher priority fragment constraint. Note that it is also possible for inconsistent constraint sets to be created by taking the union of conflicting constraints of the same priority. For example, constraining the same edge to be both in and out of an assembly or the same two fragments to be oriented in both the same and opposite directions are inconsistencies. If an inconsistency is detected in `fa_init_assemble`, a user defined warning handler will be called, and the assembly will proceed without using the most recently added constraint which caused the inconsistency to be discovered. Also note that when a constraint is created or evaluated by another FAKII function, references to fragments are to the active member of the class containing it.

For certain types of edge IN constraints, it is necessary to include containing edges and transitive edges in the constraint so that they are assembled properly. Each edge IN constraint is now "expanded" to include consistent related edges. In any constraint conflicts involving the expanded constraints, both the original and the included constraints are reported. The ability to correctly expand constraints depends on the use of an appropriate error limit with fa_add_class/frag and suitable values of asm_error_rate and asm_distrib_thresh being passed to fa_init_assemble(). If, for example, the error limit or asm_error_rate is too low, edges needed for constraint expansion will not be found and constraint conflict errors will be reported.

Function `fac_all_frags` returns a reference to a constraint set that asserts that all active fragments are to be considered in the assembly. If two constraint sets are merged using the `fac_union` function and either set contains the "`fac_all_frags`" constraint, the resulting constraint set will assert that all fragments, except those referenced in "`fac_frag_out`" constraints in the merged set, will be considered in the assembly.

A constraint set built by `fac_frag_in` asserts that the specified fragment is to be among those assembled. Function `fac_frag_out` creates a constraint set that asserts that the specified fragment is not included in the assembly. Thus one may specify a set of fragments to be assembled either by listing which ones are in, or by listing which ones are out. In the first case, one builds a constraint set of "`fac_frag_in`"s, and in the latter case one builds a constraint set of "`fac_all_frags`" and "`fac_frag_out`"s.

Function `fac_edge_in` returns a reference to a constraint set that asserts that one of the edges (if any) between `frag1` and `frag2` meeting the the orientation and overlap conditions imposed by `orient`, `coord1`, and `coord2` will be used to overlap the fragments in the resulting assembly. The `orient`, `coord1`, and `coord2` parameters optionally specify the relative orientation and the overlap of the edges to be considered. As in `fa_compare`, one may use a zero value for all three of these parameters, specify a non-zero value for just the orientation, or pass non-zero values for all three parameters, in each case specifying a progressively more restricted set of edges to consider.

Function `fac_edge_out` returns a pointer to a constraint set that asserts that all edges between `frag1` and `frag2` satisfying the orientation and overlap conditions (if in effect), will be disregarded while building assemblies over the graph associated with this constraint set.

Function `fac_orient_same` creates a constraint set that asserts that the two fragments specified will be in the same orientation in the resulting assembly.

A constraint set created by `fac_orient_opp` asserts that the two fragments specified will be in the opposite (or reverse complement) orientation in the resulting assembly.

Function `fac_distance` returns a reference to a constraint set that asserts that position `anchor1` with respect to `frag1` and position `anchor2` with respect to `frag2` are to be separated by at least `mingap` characters and at most `maxgap` characters in the resulting assembly. The fragments may be in either orientation with respect to their anchors and the anchors do not necessarily have to be positions in the fragment, e.g. an anchor value of −10 specifies a position 10 characters to the left of the first character of the relevant fragment.

```
FA_CSET *fac_union(FA_CSET *cset1, FA_CSET *cset2);
FA_CSET *fac_copy(FA_CSET *cset);
void     fac_destroy(FA_CSET *cset);
```

Function `fac_union` returns a reference to the constraint set that results from merging two specified constraint sets. Merging constraint sets consumes the references to `cset1` and `cset2`. To retain a reference to either of these constraint sets, `fac_copy` must be called on the reference.

Function `fac_copy` creates a new reference to a constraint set and returns the newly created reference. Procedure `fac_destroy` consumes a reference to a constraint set and frees the memory associated with the constraint set if this was the last reference to the object.

As implied by the naming of pointers to constraint sets as references, a reference counter mechanism is used to manipulate constraint sets. We have found that the conventions described above are very flexible and are best illustrated with an example.

```
FA_CSET *cf, *ce, *ca;
FA_NAME f1, f2, f3, f[n+1];

cf = fac_all_frags();
for (i = 1; i <= n; i++)
  cf = fac_union(cf,fac_frag_out(f[i]);
ce = fac_union(fac_union(fac_edge_in(f1,f2),
                         fac_edge_in(f1,f3)),
                         fac_edge_in(f2,f3));
ca = fac_union(cf,ce);
fac_destroy(ca);
```

`cf` becomes a reference to a constraint set specifying that all fragments except `f[1..n]` should be assembled. Within the loop, `fac_union` consumes the reference to `cf` and that returned by `fac_frag_out` and returns a new one to an object modeling their union. Later, when `fac_destroy` is called on `ca`, all the objects created are destroyed. If instead one had set "`ca = fac_union(fac_copy(cf),ce)`", then after the code is executed, `cf` will still be a valid reference, but all constraints associated with `ca` and `ce` will have disappeared.

From another point of view, the fundamental constraint set primitives and `fac_union` return pointers to objects whose reference count is 1. Moreover, `fac_union` does not modify the reference counts of its operands but since it needs to point at them, it effectively consumes one of the counts. `fac_copy` increments the reference count. `fac_destroy` decrements the reference count and if it becomes zero, recursively garbage collects all objects that become unreferenced as a result.

```
void     fac_write(FA_CSET *cset, FILE *stream);
FA_CSET *fac_read(FILE *stream);
void     fac_ascii_write(FA_CSET *cset, FILE *stream);
FA_CSET *fac_ascii_read(FILE *stream);
```

Procedure `fac_write` writes a constraint set to a file. The file must be open for writing. Function `fac_read` returns a reference to a constraint set read from a file. The file must be open for reading.

Procedure `fac_ascii_write` writes an ASCII representation of a constraint set to a file. The file must be open for writing. The first line of the ASCII constraint set file has the format "C(verification code): allfragin flag". Each constraint type is listed and the number of constraints of that type is given, followed by a list of the actual constraints. FA_NAMEs of fragments specified in fragment constraints are listed 10 per line. Edge constraints are listed one per line, including the FA_NAMEs of the fragments in the edge, their relative orientation if specified in the constraint, and overlap coordinates if specified. Orientation constraints are listed one per line, consisting of two FA_NAMEs and the relative orientation in the constraint. Distance constraints are written one per line, as two FA_NAMEs, two anchor positions, and the mingap and maxgap values. Function `fac_ascii_read` returns a reference to a constraint set read from an ASCII constraint set file. The file must be open for reading.

5. Generating Assemblies

The FAKII assembly generator includes the layout phase and the multi-alignment phase. The layout phase uses a greedy algorithm that respects any specified constraints, as described above. For the multi-alignment phase, we proceed by producing an initial alignment consistent with all the pairwise alignments of the edges in the layout of the previous phase. This is always possible, computationally efficient, and since the error rate is typically less than 10% produces a very good first approximation. As an improvement, a "window" is swept over this initial alignment to optimize the alignment in subregions where the use of global overlap alignments produced locally nonoptimal subalignments. Within the window, the alignment is again the result of merging pairwise alignments, but in this case, in a potentially different order according to the best pairwise alignments between the subsequences within the window. With this window-sweep we empirically find the resulting multi-alignment to be almost-everywhere optimal, especially when the error rate is less than 5%. Most complaints about current fragment assembly software are due to suboptimal results in the overlap and multi-alignment phases. We thus believe it is imperative to use the best possible alignment algorithms in these phases.

The following functions can be used to generate assemblies from an overlap graph:

```
void  fa_init_assemble (FA_GRAPH *graph, FA_CSET *cset, void (*handler()),
                        double asm_ov_thresh, double asm_error_rate,
                        double asm_distrib_thresh);
      void handler (int errcode, char *warn_str);

FA_ASSEMBLY *fa_gen_assembly (FA_GRAPH *graph, int refine, int reseed);
void         fa_destroy_assembly(FA_ASSEMBLY *asm);

void  fa_finis_assemble (FA_GRAPH *graph);
```

Procedure `fa_init_assemble` prepares for the computation of assemblies over a subset of edges from an overlap graph, using a constraint set.

In practice, we have found that with FAKII version 4.2 it is useful to pass a high error limit to `fa_init_assemble`, e.g. 0.20. This is because in the overlap phase, error_limit is not an absolute "threshold", but only a "limit": the kernel guarantees to find all overlaps inside the error limit (and distribution limit), but may report additional edges as well. The inclusion of these additional edges may result in fewer contigs in the assembly phase. In some situations it may be useful to specify a lower overlap threshold, higher error limit, and lower error-distribution limit when adding fragments to an overlap graph, then vary the stringency of these values when generating assemblies. This strategy has the effect of being generous in terms of including overlaps in the graph, then being more selective at the assembly stage. Since the overlap computation takes more time than generating the assemblies, it may be convenient to include all possibly useful overlaps in the overlap graph, then "turn the knobs" later to select the desired set of edges.

The subset of edges to be considered in assemblies is determined by `cset` and by the values of the `asm_ov_thresh`, `asm_error_rate`, and `asm_distrib_thresh` parameters. If `asm_ov_thresh` is greater than 0.0, only those edges with an overlap score greater than or equal to `asm_ov_thresh` will be considered in assemblies (unless the edges have been constrained to be IN). If `asm_distrib_thresh` is greater than 0.0, then for each edge with a sufficient overlap score, an error-distribution score is computed based on `asm_error_rate`. If this error-distribution score is less than `asm_distrib_thresh`, the edge will be excluded from any assemblies. If `asm_ov_thresh` is less than or equal to 0.0, all edges in the overlap graph are eligible for assembly (except any edges constrained OUT). If `asm_distrib_thresh` is less than or equal to 0.0, no assembly error-distribution screening is done.

A call to `fa_init_assemble` also associates a user defined warning message handler with the graph. The warning handler is called by FAKII procedures operating on a graph whenever an error is encountered that can be safely ignored. For example, if inconsistencies in a constraint set are detected, the warning handler is called, and if control is returned to the Fragment Assembly Kernel, the offending constraint is ignored and the assembly process continues. If a NULL pointer is passed as the warning handler, the warning message will be sent to stderr, and the assembly process will continue.

The `fa_init_assemble` routine must be called before the first call to `fa_gen_assembly`. The graph is locked by `fa_init_assemble`, and remains locked until `fa_finis_assemble` is called. Locking prevents any changes to the graph (such as addition or deletion of fragments or edges, or changing the active fragment of a class) while assembly generation is in progress. Modifications to the constraint set associated with a graph are effectively ignored while the graph is locked, since constraint set evaluation takes place only when the constraint set gets associated with the graph in `fa_init_assemble`. Passing a locked graph to `fa_init_assemble`, `fa_write_graph`, or any FAKII routine that modifies a graph causes an error trap as described at the start of the document.

Function `fa_gen_assembly` generates the next best fragment assembly over a graph, using the constraint set specified in the call to `fa_init_assemble`. A pointer to an object of type `FA_ASSEMBLY` is returned, or the null pointer if there is no next best assembly. The same constraint set is used for each assembly; to use a different constraint set, `fa_finis_assemble` must be called, followed by a call to `fa_init_assemble` with the new constraint set. If `fa_gen_assembly` is called on a graph that has not had assembly initialized via a call to `fa_init_assemble`, the error is handled as described in the Initialization and Error Handling section. The refine parameter controls whether a refinement pass is applied to the multi-alignment. For best results, set refine to 1. The reseed parameter specifies whether a seed edge is to be added to the assembly for the purpose of identifying potential alternate assemblies. See `fa_assembly_seed` for further explanation. For typical assembly operations, reseed should be set to 1. In the case that edge IN constraints are being used to generate an exact assembly in which no additional edges are desired, reseed may be set to 0.

Procedure `fa_destroy_assembly` frees the data structures associated with an assembly.

Procedure `fa_finis_assemble` terminates assembly generation over a graph, frees the associated data structures, and disassociates the constraint set specified in the call to `fa_init_assemble` from the graph.

6. Accessing information about assemblies

```
int      fa_num_contigs(FA_ASSEMBLY *asm);
int      fa_contig_height(FA_ASSEMBLY *asm, int ctg);
int      fa_contig_width(FA_ASSEMBLY *asm, int ctg);
double   fa_contig_score(FA_ASSEMBLY *asm, int ctg);

int      fa_contig_view(FA_ASSEMBLY *asm, int ctg, int beg_col, int end_col,
                        int consensus, void (*handler)());
         void handler(int row, int col, int frag, int len, int pos);

int      fa_num_fragments(FA_ASSEMBLY *asm, int ctg);
```

```
int      fa_frag_extid(FA_ASSEMBLY *assemb, int ctg, int frag);
char    *fa_frag_sequence(FA_ASSEMBLY *assemb, int ctg, int frag);
FA_NAME  fa_frag_id(FA_ASSEMBLY *asm, int ctg, int frag);
void     fa_frag_loc(FA_ASSEMBLY *asm, int ctg, int frag,
                     int *row, int *bcol, int *ecol);
char    *fa_frag_eseq(FA_ASSEMBLY *asm, int ctg, int frag);
int      fa_frag_overlap(FA_ASSEMBLY *asm, int ctg, int frag);
int      fa_frag_orient(FA_ASSEMBLY *asm, int ctg, int frag);
FA_NAME  fa_frag_pred(FA_ASSEMBLY *assemb, int ctg, int frag,
                      int *i, int *j, int *orient);

float    fa_align_trace(FA_ASSEMBLY *asm, int ctg, int frag, int spot);
float    fa_align_frag(FA_ASSEMBLY *asm, int ctg, int frag, int spot);
int     *fa_get_trace(FA_ASSEMBLY *asm, int ctg, int frag);

FA_CSET *fa_assembly_seed(FA_ASSEMBLY *asm);
```

Function `fa_num_contigs` returns the number of contigs in an assembly. The remaining assembly information routines require a contig index as an input parameter. This index provides the user a means to iterate through the contigs in an assembly. Contigs are indexed from zero to the number of contigs - 1.

Function `fa_contig_height` returns the number of rows needed to display an assembly contig.

Function `fa_contig_width` returns the length in characters of an assembly contig.

Function `fa_contig_score` returns the score of an assembly contig.

Function `fa_contig_view` produces the portion of an assembly contig that is delineated by `beg_col` and `end_col`, and calls a user-supplied routine, `handler`, once for each fragment that lies within this window. If the `consensus` argument is non-zero, a consensus row will be computed and passed to the handler routine after the last fragment in the window. The `fa_contig_view` function returns a non-zero result if no errors occur while producing the contig rows. If an error is detected, such as a column number out of range, `fa_contig_view` returns zero. The handler routine must accept arguments specifying the row being passed, the column of the multi-alignment in which the fragment begins, the index of the fragment in the contig (or −1 if the consensus row is being passed in), the length in characters of the portion of the fragment in the window (including dashes), and the first column position of the fragment that is within the window. Note that more than one fragment can occupy the same row, in which case the handler is called more than once with the same row number. Also, `beg_col` can specify a column before the start of a particular fragment, so that the corresponding value of `pos` passed to the handler is 1, or `beg_col` can intersect a fragment so that `pos` represents the column position of the intersection point. Fragments within a contig are indexed from zero to the number of fragments - 1. For example, if fa_contig_view is called on the following 3-fragment contig with `beg_col` 9 and `end_col` 36,

```
                  beg_col                  end_col
                     |___________________________|
                     |                           |
       fragA  ccgcg-tatatga-gctcgaaagctagtaaccag-gagccggc
           fragB  ggtata--acgctcga    fragC cc-gagagc-ggctaacttagg
                     |                           |
                     |_________ window __________|
```

then `handler` will be called 3 times:

handler(1, 9, 0, 28, 9); for row 1, col 9, fragA, len 28, pos 9
handler(2, 9, 1, 12, 5); for row 2, col 9, fragB, len 12, pos 5
handler(2, 31, 2, 6, 1); for row 2, col 31, fragC, len 6, pos 1

Function fa_num_fragments returns the number of fragments in an assembly contig. As with contigs, it is assumed that the user will look through the fragments by iterating a fragment index from zero to the number of fragments - 1 and passing this index to FAKII functions.

The fa_frag_extid function returns the integer external id for the fragment with index frag in the assembly contig indexed by ctg, or -1 if no fragment with that index is found.

Function fa_frag_sequence returns a pointer to the original unedited sequence of the fragment with index frag in the assembly contig indexed by ctg, or NULL if no fragment with that index is found.

Function fa_frag_id returns the FA_NAME for the fragment with index frag in the assembly contig indexed by ctg. Procedure fa_frag_loc passes back the row, *row, and beginning and ending column positions, *bcol and *ecol, of the fragment indexed by frag in the assembly contig indexed by ctg.

The function fa_frag_eseq returns a pointer to a character buffer containing the aligned sequence for the fragment with index frag in the assembly contig indexed by ctg, or if frag is −1, the consensus sequence is returned. The aligned sequence for a fragment is the sequence of characters (including dashes) representing the fragment in the multi-alignment. The buffer containing the aligned sequence is overwritten each time fa_frag_eseq is called. Function fa_frag_overlap returns FA_CONTAIN, FA_DOVETAIL, or 0 to indicate whether the fragment indexed by frag in assembly contig ctg is contained in another fragment, dovetailed with another fragment, or is the root (first fragment) of the contig. Function fa_frag_orient returns 1 if the fragment indexed by frag in the assembly contig with index ctg is reverse-complemented, 0 otherwise. The fa_frag_pred routine returns the orientation and edge script coordinates of the edge between frag and its predecessor in the chain of edges that make up a contig. If frag is at the root of the chain, -1 is returned.

The function fa_align_trace returns the location in the trace corresponding with the basepair in the fragment specified by spot. spot gives the location number (not counting dashes) in the fragment of the desired basepair (e.g. spot = 5 implies we want the 5th basepair from the left as oriented in the specified assembly). fa_align_trace will return a fractional number if the desired basepair corresponds to a location between calls in the trace (e.g. if fa_align_trace returns the value 3.25 then the specified basepair in the fragment corresponds to a location one fourth of the way between the third and fourth calls of the trace.)

The function fa_align_frag is exactly the same as fa_align_trace except that it takes a location in the trace and returns the corresponding location in the fragment.

The function fa_get_trace returns the trace edit script of the desired fragment within the assembly. It should be used in correlation with fa_add_frag when one wishes to add the edited fragment associated with the assembly as a new version of the fragment within the graph.

Function fa_assembly_seed returns a reference to a constraint set that contains a "fac_edge_in" constraint for the seed edge used to generate the assembly. The seed edge selected for an assembly is the highest scoring edge that is not constrained (by a "fac_edge_in" constraint) to be in the assemblies, and that has not yet been included in a previously generated assembly over the relevant graph. This selection of the seed edge is intended to give rise to alternate assemblies.

The seed edge can be used to regenerate its assembly without having to produce any of the preceding assemblies. For example, suppose that 10 assemblies have been generated over *graph1* with associated constraint set *cset1*. If the seed for assembly number 10 has been saved, then this assembly can be reproduced after all of the assemblies have been destroyed and fa_finis_assemble has been called as follows:

```
FA_GRAPH    *graph1;
FA_CSET     *cset1, *seed_10;
FA_ASSEMBLY *asmb;

cset1 = fac_union(cset1, seed_10);
fa_init_assemble(graph1, cset1);
asmb = fa_gen_assembly(graph1, TRUE, TRUE);
```

This approach saves the time required to generate the first 9 assemblies. Note that if the function fa_assembly_seed is called with asmb, the seed returned is not likely to be the same as seed_10, since seed_10 was added to cset1 which prevents it from being selected as a seed unless there are no unused edges.

```
void         fa_write_assembly(FA_ASSEMBLY *asm, FILE *stream);
FA_ASSEMBLY *fa_read_assembly(FA_GRAPH *graph, FILE *stream);
void         fa_ascii_write_assembly(FA_ASSEMBLY *asm, FILE *stream);
FA_ASSEMBLY *fa_ascii_read_assembly(FA_GRAPH *graph, FILE *stream);
```

Procedure `fa_write_assembly` writes an assembly to a file. The file must be open for writing, and the graph from which the assembly was generated must be saved independently by calling `fa_write_graph`. Function `fa_read_assembly` reads an assembly in terms of an existing graph. The file from which the assembly is to be read must be open for reading.

Procedure `fa_ascii_write_assembly` stores an ASCII representation of an assembly in a file. The specified file must be opened for writing before `fa_ascii_write_assembly` is called. The graph from which the assembly was generated must be saved independently by calling `fa_ascii_write_graph`. The first line of the ASCII assembly file has the format "A(verification code): number of contigs in assembly". The second line contains the assembly seed edge representation. The assembly representation is divided into three groups. In the first group, for each contig in the assembly the number of fragments, rows, columns, score, and edit information is given. For each fragment in the contig, layout information is stored. The consensus for the contig is written to the file, followed by internal consensus and layout information. The second group lists internal information for fragments in each row of each contig, and the third group contains more internal layout and edit information. Function `fa_ascii_read_assembly` reads a previously stored ASCII representation of an assembly from a file that has been opened for reading and returns a pointer to the assembly. The graph from which the assembly was generated must be extant when `fa_ascii_read_assembly` is called.

7. Editing multi-alignments

```
void  fa_swap_rows(FA_ASSEMBLY *asm, int ctg, int row1, int row2);
char *fa_get_col(FA_ASSEMBLY *asm, int ctg, int col);
void  fa_delete_col(FA_ASSEMBLY *asm, int ctg, int col);
void  fa_insert_col(FA_ASSEMBLY *asm, int ctg, int col, char *seq);
void  fa_substitute_col(FA_ASSEMBLY *asm, int ctg, int col, char *seq);
void  fa_undo_edit(FA_ASSEMBLY *asm, int ctg);
```

The multi-alignment edit functions operate on the assembly `asm`, in the contig indexed by `ctg`. Procedure `fa_swap_rows` swaps `row1` and `row2`.

Procedure `fa_get_col` returns a null-terminated string containing the symbols from column `col`. The string is overwritten each time `fa_get_col` is called.

Procedure `fa_delete_col` deletes the column specified by `col`.

Procedure `fa_insert_col` inserts the column of characters specified by the null-terminated string `seq` before column `col`. The length of the string `seq` must be the same as the number of rows in the contig. Valid characters for `seq` include dashes and characters representing an encoded nucleotide set.

Procedure `fa_substitute_col` replaces the the specified column in the multi-alignment with the column passed in via `seq`. The `seq` string must be NULL-terminated, and must be equal in length to the number of rows in the contig indexed by `ctg`, which is returned by the `fa_contig_height` function.

Procedure `fa_undo_edit` reverses the last edit made to the assembly contig.

As an illustration of the use of the editing functions, consider the multi-alignment from contig 0 of assembly `asmb`:

```
     fragA:      ggctaccgc-ctac
     fragB:          accgcggta-gga
     fragC:               g-tacggaaca

 Consensus:      GGCTACGGCG?TACGGAACA
```

The consensus column 11 could not be determined from the given fragment sequences. If the user determines that the seventh base in FragB is actually a ’c’ rather than a ’g’, the following FAKII calls can be used to correct the multi-alignment:

```
fa_delete_col(asmb, 0, 11);
```

```
fa_insert_col(asmb, 0, 11, "cc-");
```

8. Building and Using the Fragment Assembly Kernel

The Fragment Assembly Kernel package is comprised of several C source and header files, and an accompanying Makefile. The Unix "make" command can be used to build the FAKII source files into a library that can be linked with a user program. The user program must contain the directive '#include "fa_interface.h"'. The files "fa_interface.h" and "fa_errors.h" must be accessible through the include path for the user program, or may simply be placed in the directory in which the user program resides.

In addition, the Fragment Assembly Kernel uses a score table file that contains very large tables. The file is named with a ".i" extension, and by default has been built for the Sun4 platform. For other architectures, the "make all" command can be used to regenerate the FAKII score table file. This file may be installed in any directory, provided that the pathname of this directory is passed to `fa_startup`. To avoid hardcoding this pathname into the user program, the standard C library routine *getenv* can be used to check an environment variable that can be set to the pathname. If the path parameter passed to `fa_startup` is a NULL pointer or an empty string(""), the Fragment Assembly Kernel will attempt to find the FAKII score table file in the current directory.

The maximum length of fragments assembled by FAKII is dependent on the constant LENMAX, defined in the file fa_global.h. If the error FA_ERR_SCORE_OVERFLOW occurs (or the message "Error: score table limits exceeded" appears), LENMAX should be increased, and the score table rebuilt by using the "make all" command. The value of LENMAX should be approximately 1.6 times the length of the longest fragment to be assembled by FAKII. If LENMAX is made larger, the constant DIFFMAX may also need to be increased. DIFFMAX must be greater than or equal to 2 * max_error_limit * LENMAX, where max_error_limit is the maximum error rate for overlaps.

The speed of overlap computations is affected by the size of the index structure created. This size is determined by the defined constant OVTUPLE. The memory requirement for the index is sizeof(int) * pow(4, OVTUPLE). In other words, each time OVTUPLE is increased by 1, the memory requirement grows by a factor of 4. For example, if OVTUPLE is 10 and the size of an integer is 4 bytes, the index uses 4Mb of memory. For a given value of OVTUPLE, the fastest overlap computation will result when the error rate is less than 0.5/OVTUPLE. The second most efficient overlap computation will occur with error rates between 0.5/OVTUPLE and 1.0/OVTUPLE. Higher error rates may be specified, which will increase the time required to compute the overlaps.

9. Conclusion

We have produced a fragment assembly tool that is flexible and robust, yet efficient. FAKII users may choose between completely automatic assembly and a high degree of user control. Our Fragment Assembly Kernel consists of what we feel is the simplest possible set of atomic yet sufficient primitives to support proven methods of fragment assembly as well as new sequencing techniques.

In our implementation we have strictly maintained the objected-oriented paradigm: the kernel realizes objects of type *overlap graph*, *constraint set*, and *assembly* that may be created, destroyed, and manipulated only via routines of the kernel. An object persists until it is explicitly destroyed.

The kernel developed actually represents the Arizona group's second such construction effort [Kec91,MiM91,KeM93]. This second effort started from scratch with a complete redesign of the underlying algorithms and interface.

10. FAKtory Additions (Internal Only)

To facilitate the prescreening of fragments with respect to vector and other sequences, the overlap routine `fa_overlap_seqs` has been added:

```
int *fa_overlap_seqs(char *seq1, char *seq2,
                     double erate, double *score, double *erlang);
```

The two strings are compared to each other in both orientations in search of an overlap within the constraints of the three overlap parameters. A null pointer is returned if no such overlap exists, otherwise one gets a pointer to a script array which treats `seq1` as the `A` sequences, and `seq2` as the `B` sequence. Note carefully, that the script array returned will be guaranteed to remain unchanged only until the next kernel call. Furthermore the kernel will manage the space it occupies, a user should never free it. Also observe carefully that the parameters `score` and `erlang` are used for both input and output. On input they threshold the type of edges searched for, on output they are the probabilistic and distributional scores of the overlap found (if any).

As a convenience, one can now manage a pointer associated with every fragment/`FA_NAME` in an overlap graph.

```
void  fa_set_auxptr(FA_GRAPh *graph, FA_NAME name, void *auxptr);
void *fa_get_auxptr(FA_GRAPh *graph, FA_NAME name);
```

The routine `fa_set_auxptr` sets the pointer for fragment `name` in graph `graph` to the value `auxptr`. The routine `fa_get_auxptr` returns the current value of the pointer. Note that the value of this pointer is not read or written by the I/O routines and thus must be reset whenever a graph is newly input. This is the only thing that generally makes sense anyway as whatever is pointed to has invariably been moved as well in such circumstances.

To aid an application in trying to save edited regions of an assembly , the routine `fa_conform` has been added.

```
int *fa_conform(FA_ASSEMBLY *assem1, int ctg1, FA_ASSEMBLY *assem2,
                int ctg2, int c1, int c2);
```

This routine defines its source region to be the section of assembly `assem2`, contig `ctg2` between columns `c1` and `c2`. Its target assembly is assembly `assem1`, contig `ctg1`. `fa_conform` finds the section of the target assembly which corresponds to the source region and edits that section where possible so that it matches the source region. `fa_conform` assumes that the entire sequence of all fragments in the source assembly which overlap the source region exactly matches the sequence of some fragment in the target assembly. The procedure will only make edits that result in columns which look exactly like columns (after some perturbation of the rows) of the source. Thus due to new fragments in the target assembly, the portion of the target made to look like the source may be broken into sections. The first element of the array returned by `fa_conform` contains the number of sections made to look like the source, followed by the starting and ending column numbers of those sections.

Finally, contraints have been extended to include a status of "may" or "must" and in-edge constraints may now be specified in a way that is more natural for users. For every ‘‘`fac`’’ routine, not dealing with fragments, there is now also a corresponding ‘‘`fam`’’ routine that is identical save for an additional integer parameter `must` inserted right after the two fragment parameters. This parameter may be given as the constant `FAM_MUST` or `FAM_MAY` depending on whether the constraint is a must or may constraint. All ‘‘`fac`’’ constraints are assumed to be must constraints.

```
#define FAM_MUST 0
#define FAM_MAY  1

#define FAM_EXACT      0x000
#define FAM_OFFSET     0x002
#define FAM_OVERLAP    0x004

#define FAM_ANY        0x000
#define FAM_DOVETAIL   0x008
#define FAM_CONTAIN    0x010

#define FAM_ORDERED    0x020

#define FAM_SWAP       0x040

#define FAM_REVERSE    0x080

FA_CSET *fam_edge_in    (FA_NAME frag1, FA_NAME frag2, int must_flags,
                         int orient, int coordA, int coordB);
FA_CSET *fam_edge_out   (FA_NAME frag1, FA_NAME frag2, int must,
                         int orient, int coordA, int coordB);
```

```
FA_CSET *fam_orient_same(FA_NAME frag1, FA_NAME frag2, int must);
FA_CSET *fam_orient_opp (FA_NAME frag1, FA_NAME frag2, int must);
FA_CSET *fam_distance   (FA_NAME frag1, FA_NAME frag2, int must,
                         int anchor1, int anchor2,
                         int mingap1, int mingap2);
```

In addition, the must-argument of `fam_edge_in` may involve the bitwise-or of one of `FAM_EXACT`, `FAM_OFFSET`, or `FAM_OVERLAP`, one of `FAM_ANY`, `FAM_DOVETAIL`, or `FAM_CONTAIN`, and optionally any of `FAM_ORDERED`, `FAM_SWAP`, and `FAM_REVERSE`. If `FAM_EXACT` is set, then all other bits are ignored and the overlap is as for the corresponding `fac` routine. Otherwise the specification is of an overlap as in the FAKtory constraint class panel where each bit setting corresponds to the appropriate button on the panel. In addition the parameter `orient` is either 0, `FA_COMP`, or `FA_SAME` according to whether or not an orientation constraint is also being given, and the two `coord` parameters specify the low and high end of the range for the specified overlap entity.

11. References

[Hua92] Huang, X. "A contig assembly program based on sensitive detection of fragment overlaps". *Genomics* **14** (1992), 18-25.

[Kec91] Kececioglu, J.D. "Exact and approximate algorithms for DNA sequence reconstruction". Ph.D. Thesis. Technical Report 91-26, Dept. of Computer Science, U. of Arizona, Tucson, AZ 85721.

[KeM93] Kececioglu, J.D. and E.W. Myers. "Combinatorial algorithms for DNA sequence assembly". Accepted for publication in *Algorithmica* (1993).

[MiM91] Miller, S. and E.W. Myers. "A fragment assembly project environment". Technical Report 91-17, Dept. of Computer Science, U. of Arizona, Tucson, AZ 85721.

[PSU84] Peltola, H., H. Söderlund, and E. Ukkonen. "SEQAID: A DNA sequence assembly program based on a mathematical model". *Nuc. Acids Res.* **12** (1984), 307-321.

[Sta82] Staden, R. "Automation of the computer handling of gel reading data produced by the shotgun method of DNA sequence". *Nuc. Acids Res.* **10** (1982), 4731-4751.