The GAF Data Exchange Format

Gene Myers

TR 99-02

The GAF Data
Exchange Format

Gene Myers

TR 99-02

ABSTRACT

GAF is a General Assembly Format for exchanging data in a sequencing project. It provides constructs for specifying sequences, quality numbers, assemblies, multi-alignments, overlaps between fragments, and constraints between them. It is designed for coding the information at any point in the assembly process, so for example, a GAF file can specify just a collection of fragments, a collection of fragments and assigned quality numbers, a collection of fragments and their overlaps, an assembly of some fragments, assemblies of assemblies of fragments, and so on. Its specific purpose is to facilitate the exchange of data between a suite of cooperative software processes handling different aspects of an overall sequencing informatics pipeline. A very simple translation (using a simple awk script) can convert the CAF format introduced by the Sanger group into a subset of the GAF format.

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.

The GAF Data Exchange Format

Two basic principles were followed in devising the GAF format: (1) input should be easy in that it should be trivial to parse, and the information should arrive in a sequence suitable for the creation of the data-structures needed by the process, and (2) output should be easy in that one should simply be able to write the information in whatever order it occurs in the data structures of the application in question. To accomplish this goal we adopted a simple ASCI line-based format with an html-style syntax in anticipation of a possible web-publication connection. With respect to the order of information we have adopted the idea that there will be a suite of GAF Utilities that will not only automatically reorder the information, but also validate various aspects of the data set, and further compress and decompress GAF files for storage.

The GAF format was based on a series of discussions between Granger Sutton, Xiaoqui Huang, and myself while at TIGR. All three of us have built and continue to refine assemblers and it was our desire to be able to mix-and-match various components of our systems based on the realization that our respective systems had both strengths and weaknesses, so that the best system would be the best parts of our respective systems. Moreover, such an exchange format would permit us to share components and focus on building great individual components.

General and Grammatic Conventions:

We will present the GAF format by interspersing textual description with formal context-free grammar specification of the language. The conventions for our grammatical specification are as follows.

- Productions are written as: `N <- expr`, where `N` is a non-terminal and expr is a regular expression of non-terminals and terminal strings.
- A non-terminal is a descriptive word between angle-brackets or square brackets, e.g., `<contig>` or `[info]`. Those non-terminals within square brackets must occupy one line of the input.
- Terminal strings are distinguished by occurring between double-quotes and may be a regular expression formatted according to egrep conventions (a UNIX standard), in which case the terminal string being specified is any matching the regular expression. For example, `"[acgtACGT]+"` matches any non-empty string of upper or lower case DNA sequence letters.
- The right-hand side of a production may be any regular expression of non-terminals and terminal strings built up with the egrep operators |, (concate), ?, *, and +. Moreover, spaces between concatenated terms implies that white space (space and tab) must exist between the items involved, and conversely, no space between terms implies no white space exists between the instances of the items involved. For example, `"begin" <id>"("<int>")" * "end"`, specifies a list of identifiers each followed by a parentheses bracketed integer and delimited by the strings `begin` and `end`. There must be white space between the bracketing strings and each item in the list, but not between the identifier and its parentheses bracketed integer.

In addition to these syntactic conventions, there are a few semantic conventions that hold throughout. Specifically,

- Each class of integer ids for the various objects, i.e. `ctg_id`, `ovl_id`, and `rel_id`, below, are assumed to be ints in a contiguous range beginning at 0.
- Dashes in sequences do not count as index positions.
- Indices are always in absolute coordinates (as opposed to "clipped" coordinates). This only affects index computations when a sequence represents a clipped sequence whose clipping interval is given.

The GAF specification language borrows the style of the html language, in that key-symbols consist of an identifier with angle braces, e.g. `<info>`, and that a multi-line object whose description begins with a key-symbol `<X>`, is terminated with a bracketing `</X>`, e.g. `<contig>` and `</contig>`.

The GAF language admits variability in the content of some object specifications in order to accommodate extension and customizability for groups or tool producers that wish or must so augment the language. To facilitate this, GAF permits many of its initiating key-symbols to have a parenthesized ‘‘type name’’ follow the key-name. These names may be arbitrarily chosen and consumers of a GAF-file can either ignore or utilize the implication of the presence of the type name in the data of the object. For example, our assembler, FAKII, may produce a `<contig(FAKII)>` contig so that other compatible programs, may take advantage of the fact that FAKII produces a certain subset of the variable parts of a contig specification and that these parts have a particular interpretation. Another example might be a `<info(Sizes)>` line which is guaranteed to contain the number of each type of object so that one can, for example, allocate a single array of structures for each collection.

The GAF Objects:

At the top level a GAF spec can consist of six different types of data units in any order: an information line, a contig, an overlap, a constraint, and a relation. Formally:

```
<spec> <- ( [info] | <contig> | <overlap> | [constraint] | <relation> )*
```

1. Information Lines.

An information line permits the specification of any data a user desires. It consists of an `<info>` key-symbol followed by whatever information is desired. It is in effect a comment line to any process that wishes to ignore the content of the line. The info keyword can be typed so that consumers of information can easily identify which information lines they should process and the nature of the information on the line.

```
<info> <- "<info"("("<type:var>")")?">" ".*"
```

2. Contig Objects.

In contrast to the information line, a contig object is the most proscribed GAF object and is the single vehicle by which one specifies sequence reads, assemblies, and layouts. A contig object always consists of (1) a header line and an end line, (2) either a sequence or a layout or both, and (3) some optional information specifying joins, scores, clips, and tagged regions.

```
<contig> <- [ctg_hdr]
               (<sequence> <quality>*) ? <layout> ?  # 1 or both
               (<joins> [score] [clip] [tag]) *
            [ctg_end]
```

The contig header starts with an optionally typed `<contig>` key-symbol, followed by an optional symbolic name, a mandatory contig id number, the contig’s sequence length in bases, and the padded length of the contig. A contig’s sequence length is the number of bases specified in the sequence component of contig if any. If not then it equals the pad-length of the contig. The pad-length of a contig is the number of columns in the multi-alignment, if any, represented by the contig. If the contig does not represent a multi-alignment/sequence, but just a layout, then the pad-length may be specified as per the generating agents whim. The contig trailer consists of a line with the bracketing `</contig>` key-symbol.

```
[ctg_hdr] <- "<contig"("("<type:var>")")?">"
                  <name:var>? <ctg_id:int> <length:int> <pad_len:int>
[ctg_end] <- "</contig>"
```

2.A. Sequences.

If the contig has a sequence then this is simply specified on a collection of lines between a `<sequence>` and `</sequence>` bracketing pair, each on its own line. The sequence represent either the initial bases of a sequence read, or the consensus sequence reconstruction from an assembled collection of reads. The sequence can contain dashes interspersed between its characters in which case the length of the sequence including the dashes is the contigs pad-length. These may be present in a read in order that align with other reads in a contig object representing an assembly, or they may be present in a consensus sequence in order to correctly model columns whose consensus symbol is a ‘-’.

```
<sequence> <- [seq_hdr] ([seq_lines])* [seq_end]

      [seq_hdr]   <- "<sequence>"
      [seq_line]  <- "[a-zA-Z-]*"
      [seq_end]   <- "</sequence>"
```

A sequence may be optionally followed by one or more quality sequences that are a list of numbers sandwiched between `<quality>` and `</quality>` bracketing lines. The sequence numbers must have the same length as the character sequence they qualify. The quality key-symbol may be qualified to indicate to a consumer the type of quality information being provided, e.g., `<quality(Phred)>` or `<quality(Signal2Noise)>`.

```
<quality>  <- [qual_hdr] ([qual_line])* [qual_end]

   [qual_hdr]   <- "<quality"("("<type:var>")")?">"
   [qual_line]  <-  <number> *
   [qual_end]   <- "</quality>"
```

For example a sequence read with associated Phred numbers might have a GAF entry as follows:

```
<contig> sb115.r 13 100 100
<sequence>
acgcagcaccagagtatgactagctacgatcgagca
acgctgctcgatcgaaacgatcgatcccatcgatcc
</sequence>
<quality(Phred)>
10  5 32 ...
</quality>
</contig>
```

2.B. Assemblies: Layouts and Multi-Alignments.

In the case that the contig object represents an assembly it can have a layout component. A layout consists of a `<layout>`, `</layout>` pair that, in the case of a `basic_spec`, bracket a series of lines each containing a reference to a contig subobject and the index range (in the padded interval) of the current contig that is covered by the referenced sub-contig. In other words, one gives a list of the contig objects that were assembled to form the current contig (these could be either reads or sub-assemblies), and their positions in the assembly.

```
<layout> <- [layout_hdr] <basic_spec> | <delta_spec> [layout_end]

   <basic_spec>  <- [contig_ref] *
   <delta_spec>  <- <delta> ? ( [contig_ref] <delta> )*

   [layout_hdr]  <- "<layout>"
   [layout_line] <- <ctg_id:int>?"("<int>","<int>")"
   [layout_end]  <- "</layout>"
```

If one wishes to further indicate how to align the sequences of the sub-contigs together into a proper multi-alignment, then the `delta_spec` format permits one to indicate where dashes should be interspersed between the symbols of the sub-contigs and, optionally the consensus sequence for the assembly (if one is given), so that all sequences align as desired. The positions at which dashes occur is specified by an optional list of integers between a `<delta>`, `</delta>` pair following each contig reference. If the assembly has a sequence component, then the ‘‘delta’’ for this consensus sequence is given before the first sub-contig reference. Each delta header is followed immediately by the length of the delta sequence. As an example, if the referenced contig has sequence, `acgta`, and the delta is 1 3 3 6, then the dashed counterpart would be, `-ac--gta-`.

```
<delta>  <- ["<delta>" <length:int>] ([<int> *])* ["</delta>"]
```

While at one extreme a contig object can encode simply a sequence or read, at the other it can present just a layout of a collection of clones devoid of any sequence information. We call such an arrangement without any sequence information a ‘‘layout’’. For example,

```
<contig> Layout1 10 55 55
<layout>
0(1,25)
1(11,40)
2(26,50)
3(31,55)
</layout>
```

represents the following assembly:

```
0 -------------------------
          1 ------------------------------
                          2 -------------------------
                              3 -------------------------
```

without any connection to the underlying sequence giving rise to the assembly. Thus, a given software component may decide how to assemble a collection of sequences and add to a GAF file, a contig object such as the one above, that just gives the relative positions at which to place each sequence. Presumably a later ‘‘multi-alignment’’ module would then take up the task of aligning the sequence of all the fragments so as to determine the most probable sequence reconstruction.

There are two principle ways of encoding a multi-alignment (with or without consensus sequence) under the GAF format. For illustration purposes consider the following multi-alignment:

```
ata-aagtgagaccttttcctgatagcg-tctacca-tgaaag-atgt
aca-aagtgagaccttttc-tgatagcg-tctacca-tgaa
     agtgagaccttttc-tgataggg-tccaccattgaaag-atgt
                            ctctacca-tgaaag-at
------------------------------------------------
ANA-AAGTGAGACCTTTTC-TGATAGCG-TCTACCA-TGAAAG-ATGT
```

In the first style (used in the CAF format [Dea98]), the sequences of the sub-contigs and the consensus of the assembly-contig (if there is one), have dashes interspersed in them so that when the first character of each sub-contig is aligned with the first character/position of the contig’s sequence covered.

```
<contig> Fragment0 0 44 48
  <sequence>
    ata-aagtgagaccttttcctgata
    gcg-tctacca-tgaaag-atgt
  </sequence>
</contig>

<contig> Fragment1 1  37 41
  <sequence>
    aca-aagtgagaccttttc-tgata
    gcg-tctacca-tgaa
  </sequence>
</contig>

<contig> Fragment2 2 39 42
  <sequence>
    agtgagaccttttc-tgataggg-t
    ccaccattgaaag-atgt
  </sequence>
</contig>

<contig> Fragment3 3 16 18
  <sequence>
    ctctacca-tgaaag-at
  </sequence>
</contig>

<contig> Multi1 4 43 48
  <sequence>
  ANA-AAGTGAGACCTTTTC-TGATA
  GCG-TCTACCA-TGAAAG-ATGT
  </sequence>
  <layout>
    0(1,48)
    1(1,41)
    2(6,48)
    3(29,46)
  </layout>
</contig>
```

Note carefully, that sub-contig intervals are given as the indices of columns in the multi-alignment, or as subintervals of the pad-length interval [1,48] (and not [1,43] the length of the consensus). Moreover, if one did not wish to specify the consensus (perhaps another subsequent agent will compute this), then one can simply leave out the sequence component of `Multi1` and set its contig length to 48.

While the above style is quite simple to realize, it has the disadvantage that one cannot encode several different possible assemblies of the same set of fragments, as a fragment potentially requires dashes at different positions for each assembly. The second, delta-style encoding uses deltas, specified in the assembly, to indicate where to place dashes in each reference sub-contig. Continuing with our example, one would have:

```
<contig> Fragment0 0 44 44
  <sequence>
    ataaagtgagaccttttcctgatag
    cgtctaccatgaaagatgt
  </sequence>
</contig>

<contig> Fragment1 1  37 37
  <sequence>
    acaaagtgagaccttttctgatagc
    gctctaccatgaa
  </sequence>
</contig>

<contig> Fragment2 2 39 39
  <sequence>
    agtgagaccttttctgatagggtcc
    accattgaaagatgt
  </sequence>
</contig>

<contig> Fragment3 3 16 16
  <sequence>
    ctaccatgaaagatg
  </sequence>
</contig>

<contig> Multi1 4 43 48
  <sequence>
  ANAAAGTGAGACCTTTTCTGATAGC
  GTCTACCATGAAAGATGT
  </sequence>
  <layout>
    <delta> 5
      4 19 27 34 40
    </delta>
    0(1,48)
      <delta> 4
      4 28 35 41
      </delta>
    1(1,41)
      <delta> 4
      4 19 27 34
      </delta>
    2(6,48)
      <delta> 3
      15 23 37
      </delta>
    3(29,46)
      <delta> 3
      1 9 15
      </delta>
  </layout>
</contig>
```

Note carefully that the padded length of `Multi1` is its consensus length plus the length of the delta for the consensus. The GAF tool suite will provide a utility for converting from one format to another in the event that there is only one assembly, and for recoding the indices if one wishes to remove a consensus sequence.

2.C. Additional Contig Attributes.

In addition to the core sequence and/or layout information, contig/sequence object description may contain the following additional items. First, in the case of assemblies or multi-alignments, one may which to indicate which set of fragment overlaps were used in putting the assembly together. This is indicated by an optionally-typed "join" pair bracketing a list of references to overlap records (to be described).

```
<joins>  <- [join_hdr] ([join_line])* [join_end]

   [join_hdr]   <- "<joins"("("<type:var>")")?">"
   [join_line] <- <ovl_id:int> *
   [join_end]   <- "</joins>"
```

In addition one may which to attribute a score to a contig object reflecting the quality or confidence in the accuracy of the assembly and/or sequence.

```
[score] <- "<score"("("<type>")")?">" <floating-point number>
```

Contig objects representing sequences are often time clipped at the ends to eliminate poor data. This information is given as a pair of indices delimiting the segment of the sequence to retain, inclusively. The clip record may be optionally typed to indicate, for example, the agent that performed the clip or the criterion used to determine where to clip.

```
[clip]  <- "<clip"("("<type>")")?">"  <index> <index>
```

Finally, one often wants to ‘‘tag’’ a substring of either an assembly or a sequence and one can do so with an optionally-typed tag record containing a pair of indices indicating the tagged range, inclusively. In this case the optional type might, for example, indicate what is being tagged, e.g Alu, L1, etc.

```
[tag]   <- "<tag"("("<type>")")?">"   <index> <index>
```

3. Overlap.

One of the major subtasks in assembling a collection of sequences is to determine the overlaps between them. Such overlaps are described as follows in GAF format. The usual, optionally-typed `<overlap>`, `</overlap>` pair bracket the description where the opening bracket contains an optional symbolic name for the overlap and a mandatory overlap id-number.

```
<overlap> <- [ovl_hdr] [ovl_desc] [ovl_align]? [score]* [ovl_end]

   [ovl_hdr]   <- "<overlap"("("<type:var>")")?">" <name:var>? <ovl_id:int>
   [ovl_desc]  <- "<spec>" <ctg_id:int>"("<int>,<int>")"
                                <ctg_id:int>"("<int>,<int>")"
   [ovl_align] <- ["<align>"] [ <int> * ]* ["</align>"]
   [ovl_end]   <- "</overlap>"
```

The nature of the overlap is given by specifying the two contigs involved by their id-numbers, each followed by the range of the sequence aligned with the other. Note that this actually permits the specification of locally aligned parts of the sequence as well as true end-to-end overlaps.

A precise description of an alignment between the two overlapping parts is optional, and if present is given as a list of integer positions between a `<align>`, `</align>` pair, where each index indicates the character before which to insert a dash in order to get the desired alignment between the two sequences. While this could have been done with two `<delta>` records, it is more convenient to give one integer list, where positive integers indicate dashes in the first contig and negative integers indicate dashes in the second sequence. For example, the overlap:

```
                .     .    .
        12: accgta-ctacgatacacgg
        15:     tacctactatac-cggattacag
                    ^    ^     ^    ^
```

could be encoded by the following overlap record:

```
        <overlap> 12 15
          <spec> 12(5,19) 15(1,15)
          <align>
            7 -13
          </align>
        </overlap>
```

Finally one may wish to score an overlap in one or more ways and such scores can be listed on optionally-typed `<score>` lines.

4. Relationships and Constraints.

It is often the case in an sequencing project that additional information is known about the relationship of fragments in a valid solution. The most typical case is when pairs of fragments have been sequenced from both ends of an insert. The GAF permits one to model such a constraint between a pair of sequences or contigs with a `constraint` line. Such a line indicates the ordered pair of sequences involved by id-number, the relationship between them, also by id-number, and whether the constraint must be true in a valid solution, or whether it may be violated if there is sufficient contradictory evidence.

```
    [constraint] <- "<constraint>" "must"|"may"
                        <ctg_id:int> <ctg_id:int> <rel_id:int>
```

The nature of a constraint is specified by a `relation` object which is an optionally-typed `<relation>`, `</relation>` pair bracketing the description where the opening bracket contains an optional symbolic name for the overlap and a mandatory overlap id-number. The relationship is specified as a list of relationship items, the conjunction of which constitutes the desired relationship.

```
    <relation> <- [rel_hdr] [rel_lines] [rel_end]

       [rel_hdr]    <- "<relation>" <name:var>? <rel_id:int>
       [rel_lines]  <- "same" | "opposite" | "no_overlap"
                     | "overlap" "reversed"? ("proper|contains")?
                          (("A"|"B")?"offset"|"overlap")"("<int>","<int>")" ?
                     | "distance" <anchor1:int> <anchor2:int>
                          "("<min:int>","<max:int>")"
       [rel_end]    <- "</relation>"
```

The relationship items and their interpretations are as follows:

`same`: A fragment may be used directly in a solution or its Watson-Crick complement, constituting the sequence on the opposing strand of a duplex, is used. This is called the orientation of the fragment. The keyword `same` requires that the fragment should be used in the same orientation.

`opposite`: The fragments should be in opposite orientations.

`no_overlap`: The fragments should not overlap in the solution.

`overlap`: The fragments should overlap and optionally in one or more of the following more specific way:

- `reversed`: When a constraint between two fragments is given, the first fragment is assumed to be the ‘‘A’’ fragment and the second the ‘‘B’’ fragment for the purposes of the description that are about to follow. This keyword indicates that the roles of A and B should be reversed.
- `proper`: The fragment overlap involves only proper prefixes and/or suffixes of the two fragments.
- `contains`: The B fragment is completely contained within the A fragment.
- `Aoffset(i,j)`: The A fragment has an initially unaligned segment of length between i and j inclusive.
- `Boffset(i,j)`: The B fragment has an initially unaligned segment of length between i and j inclusive.
- `overlap(i,j)`: The length of the overlap between the two sequences is between i and j inclusive.

`distance a1 a2 (min,max)`: Consider a fragment on an infinite axis whose origin is immediately to the left of its first base, and where one unit of length corresponds to one base. The two anchors *a1* and *a2* are coordinates on such an axis relative to the first and second fragments, respectively. For example, -100 is 100 bases to the left end of a fragment, +100, 100 bases to the right. This constraint specifies that in any solution, the distance between anchors *a1* and *a2* should be in the range *[min,max]* inclusive.

Note of course that some combinations of relationship items do not make sense and one should not generate such combinations in a GAF file. As a simple example, suppose that reads are sequenced off the ends of inserts known to be of length between 1000 and 5000. Then the relationship capturing such mates, would be:

```
<relation> Mates 1
opposite
distance 0 0 (1000,5000)
</relation>
```

5. References

[Dea98] Dear, S., Durbin, R., Hillier, L., Marth, G., Thierry-Mieg, J. and Mott, R. "Sequence Assembly with CAFTOOLS". **Genome Research**, *In press*, (1998).