

## Sequence Graph format (SQG) specification version 0.1

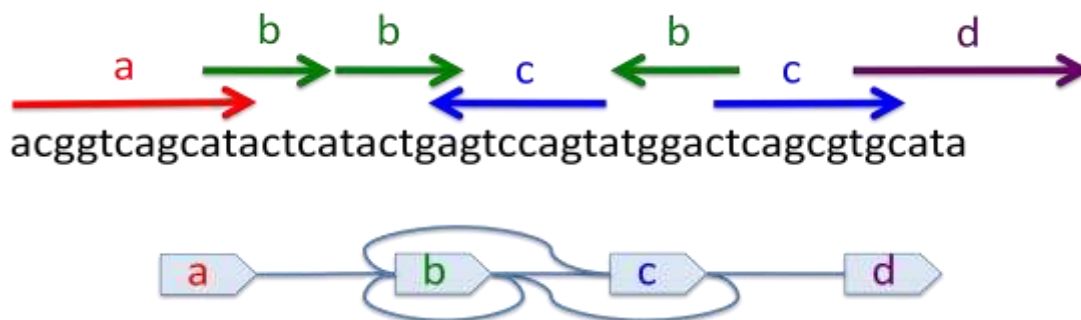
This is a straw man proposal for a standard exchange format for graphs made of connected segments of DNA. A basic implementation in C may emerge soon, depending on feedback.

SQG is designed to provide a standard for the graphs used within assembly software (assembly graphs including de Bruijn graphs and string graphs), and also to represent repeat structure in a large sequence such as a genome (repeat graphs), variation in a population, and transcript structures. The basic model is that there are two primary types of object: segments with sequence attached, and connections (adjacencies or overlaps) between the segments.

SQG supports attachment of arbitrary information to segments and connections to allow for extendability and a wide variety of uses. This is done by allowing the declaration of named properties, which are strongly typed.

SQG supports representing the consensus sequence of segments, but is not designed primarily to represent the relationship between primary sequencing reads and the segments in an assembly. We recommend use of BAM for representing the detailed alignments (for further discussion see further below).

### 1. Example and discussion of representation



SQG 0.1

HISTORY 2011/3/14\_19:04:00 made by hand

PROPERTY segment Multiplicity int number of times the node is traversed

PROPERTY connection Comment string arbitrary text

PROPERTY walk Chromosome flag the true chromosomal sequence

SEGMENT a 10 acggtcagca Multiplicity=1

SEGMENT b 6 catact Multiplicity = 3

SEGMENT c 8 ggactcta Multiplicity = 2

SEGMENT d 10 agcgtgcata Multiplicity = 1

CONNECTION a >> b -2

```

CONNECTION b >> b 0 Comment="tandem repeat"
CONNECTION b >< c -2
CONNECTION c << b 0
CONNECTION b <> c -1
CONNECTION c >> d -2
WALK 1>a>>b>>b><c<<b<>c>>d>10 Chromosome

```

In the picture for this example we have shown sequence segments as oriented blocks, and connections as non-directed edges. In terms of graph theory this is completely equivalent to two standard representations:

- The breakpoint graph, originally introduced in the context of representing the relationship between synteny blocks in chromosomes of related species. These are graphs with two types of edge: labelled segment edges, and connection edges. Any walk must alternate between segment edges and connection edges. SQG format corresponds to attaching sequence to the segments, with the nodes in the graph implicitly defined at the ends of segments. The '>' and '<' symbols in the connection lines indicate which ends of the segment edges are connected, e.g. **a>>b** means the end of **a** is connected to the start of **b** and **b><c** means that the end of **b** is connected to the end of **c**.
- Representations in which sequence is attached to nodes of a graph and there are bi-oriented edges, with rules about consecutive edges being traversed having to be "compatible". For this, SQG notation >> is the type of the edge, compatibility means that adjacent edges in a sequence walk need to end and start with the same symbol (so >> and >< are compatible, but >> and << are not), and sequence is read from nodes in the forward direction if they are flanked by >> and in the reverse complement direction if they are flanked by <<.

Which of these (or any other equivalent) is used internally by software implementing SQG is up to the software developer.

There is however an alternative natural representation of a sequence assembly as a directed graph where sequence is attached to the directed edges, and a sequence walk is just any walk following edges in the correct orientation. We believe we can map between this representation and the SQG representation, but can not guarantee to get back to the same actual version. An open question is whether we can define a canonical version that is stable.

## 2. Record types

SQG is a record based format, designed to also support a binary implementation. Records are tab delimited. The first field comes from a fixed vocabulary and determines the record type.

### SQG <major> <minor>

This record must be present, must be the first record, and must be unique. <major> and <minor> are integers denoting the major and minor versions.

## HISTORY <date\_time> <command>

These are optional records, to record the sequence of commands that created the file.

- <date\_time> is date and time string with format yyyy/mm/dd\_hh:mm:ss.
- <command> is an arbitrary string – typically the literal command line obtained with ARGV[0] or the equivalent for the language being used.

Good practice is to copy any existing lines and add one more each time the file is transformed. This self-documentation feature has proved useful in BAM and VCF files.

## PROPERTY <id> <type> <description>

Properties are used to attach extra information to segments, . Property declaration lines must precede the first use of the property.

- <id> is a legitimate C identifier, satisfying regexp `[a-zA-Z][a-zA-Z0-9_]+`. Ids starting with a capital letter are reserved for definition in this or a future version of the specification – see below for reserved property names.
- <type> is currently one of {**int**, **float**, **string**, **flag**}.
  - int values are signed integers maximum  $2^{32}-1$ , minimum  $-2^{32}$ .
  - float values are parsed using a scanf family function into a 4 byte float.
  - string values are an arbitrary length sequence of printable ascii characters enclosed in double quotes, allowing the space character, newline as `\n`, tab as `\t`, double quotes as `\` and backslash as `\\`.

*It would be attractive from the user perspective to extend the type system to support more complex constructed types, in particular fixed and variable size arrays, and ideally compound types/structures. And/or we could define special types such as “annot” for annotations with start and end coordinates in the sequence and some attached information (what type?), or “depth” for pairs of a string and a non-negative floating point depth. The advantage of such special types is that they could in principle automatically transform correctly with operations on the graph.*

- <description> is a free text description of the property. There must be some description.

We allow the same property to be used for different object types, e.g. segments, connections or walks, but the attached data will be the same.

## SEGMENT <id> <length>[ <properties>[ <sequence>]]

These define the sequence units. The sequence is optional, so as to allow a more compact representation with sequence stored elsewhere. Properties are optional because there might not be any.

- <id> is a standard C type identifier. Identifiers must be unique in a graph; there can not be more than one SEGMENT record with the same id.
- <length> is a positive integer denoting the length of the sequence. In some cases where the length is uncertain, this is presumed to be a plausible “best-guess” value.

- `<properties>` is a string of the form `<prop_id>[=<value>];<prop_id>[=<value>]]*`, where `<prop_id>` is a property id, and `<value>` is a legal value. The equals sign and value must be present unless the property has type flag, in which case they can not be present. If there are no properties, but the sequence is present, then the reserved property `None` of type flag must be used (i.e. write `None` in this field).
- `<sequence>` is the sequence of the segment in the alphabet `[acgtn]`. It is annoying to support `n`'s, but experience from other formats is that it is necessary. An application can check if `n`'s are absent to implement more efficient bitwise compression.

### CONNECTION `<seg1-id> <join> <seg2-id> <separation>[ <properties>]`

These define the connections between segments.

- `<seg1-id>` and `<seg2-id>` are segment ids. They need to have been declared already in `SEGMENT` records.
- `<join>` is one of
  - `>>` if the end of `seg1` joins the start of `seg2`
  - `><` if the end of `seg1` joins the end of `seg2`
  - `<<` if the start of `seg1` joins the end of `seg2`
  - `<>` if the start of `seg1` joins the start of `seg2`
- `<separation>` is an integer. If it has value 0 then the segments abut. If it is negative then they overlap by minus the number of bases specified. If it is positive then there is a gap between the segments of the number of bases specified. As with segment lengths, the connection separation must be present and should be set to a working "best guess" if uncertain.
- `<properties>` is a property string as for segments. In this case if there are no properties `None` does not need to be used since this is the last field (which is good because `None` is a segment property).

There can be at most one connection with each join type between each pair of segment. Note that there can be multiple connections between segments if they have different join types (e.g. the graph of a diploid sequence with an inversion `b` between segments `a` and `c` would have connections `a>>b`, `b>>c`, `a><b` and `b<>c`). It is legal for there to be a connection between a segment and itself (for example this is used for tandem repeats, or exact palindromes).

### WALK `<specification> <properties>`

Walks define sequences that can be obtained by traversing arbitrary sections of the graph, starting and ending at any base position within segments.

- `<specification>` is a string with the following syntax `<int><directed_segment>+<int>`, where `<directed_segment>` is `>node_id` or `<node_id<`, where the greater than and less than symbols in these are literals indicating whether the segment is traversed in the forwards or reverse direction. So `17>a>>b>5` starts at position 17 in segment `a`, moves forwards along `a` to its end, moves into the start of node `b` (correctly handling any gap or overlap caused by the separation of the nodes `a`

and b, which must have an edge  $a \gg b$ ) and moves forwards to base 5 of b.

Segments of a single node can be specified with e.g. **27>a>192**.

Note that the reverse complement of a walk is specified by reversing the elements, so for example the reverse complement of **17>a>b>5** is **5<b<a<17**. Also note that walks have an implicit sequence and length. We support circular sequences such as mitochondrial genomes by a flag property Circular.

*We could allow the length and sequence to be given in the format. Although redundant, this would have the advantage of having a standard format for extracting sequences of walks.*

Walks are a key concept. Basically, any true sequence in the true genome(s) represented by the graph will correspond to a walk. In particular, full chromosomes will correspond to walks. Furthermore, any specific match of another sequence to a graph will match a walk, so a walk replaces the standard concept of a triple (`<sequence>`, `<start>`, `<end>`).

A set of walks induces an arbitrary subgraph, so in particular the set of matches into a graph will generate a subgraph. I am not sure how useful that concept is.

### 3. Reserved properties

#### Segment

- **ReadStartCount** *int* How many reads start in this segment
- **CopyNumber** *int* How many times this segment is traversed in a sequence set
  - Typically the sequence set in question would be the standard chromosome set for the organism.
- **CopyNumberEstimate** *float* Estimated copy number
  - It would be nice to give a distribution with probabilities, actually likelihoods. For this we could have an array of floats, with values from 0 up to  $k$  copies, and a final value indicating  $>k$ .
- **Comment** *string*

#### Connection

- **CopyNumber** *int* How many

#### Walk

- **Circular** *flag* Denotes that the described sequence is circular
  - Require that end point is start point minus 1
- **Chromosome** *string* Claim this is a full chromosome sequence
  - *string* is name of chromosome
- **Reference** *string* Corresponds to a publicly known reference sequence
  - *string* is identifier for reference
- **Annotation** *string* This walk corresponds to an annotation

## 4. Potential operations

We can think of a number of operations.

- Format validation.
- Nx determination, e.g. N50.
- Finding sequences of walks.
- Tip removal, subject to criteria such as maximum length.
- Bubble identification and collapsing.
- Standardising overlap length. There are various reasons to standardise overlap length. Removing overlaps, so that all separations are 0 (or positive if there were any positive overlaps in the original sequence) gives a non-redundant representation of all the sequence, and can in principle simplify bubbles and other types of variation. On the other hand, increasing overlaps to a standard value k will ensure that all subsequences of the full sequence (whatever it is) are represented in the node sequences, which can be useful for matching operations.
- Transformation into graphviz dot format for visualisation and some standard graph operations. And back again if sufficient information is preserved.
- Matching an arbitrary sequence. This is probably the killer app.

## 5. Open issues/options

### 5.1 How to handle typed properties?

### 5.2 Should we have some sort of hierarchical substructure?

### 5.3 Compression, binary version of format?

### 5.4 Sorting, indexing

### 5.5 Include/import macro

Might be good for walks in particular, and in context of hierarchical structure

### 5.6 Recording information for arbitrary segment pairs

This is important for scaffolding. Different from connections because the segments are not necessarily consecutive. Can have distance and/or orientation (e.g. transcript exon matches give orientation not distance), and a type. Perhaps a new record type?

### 5.7 Add read placement – Celera people want this

Full alignments are walks. Really these are better held in BAM. But could record start points, or start point density along segments.

## 6. Comparators

- Amos (U Maryland, TIGR, Celera and others)
  - Open sources assemblers, visualisers etc.
  - <http://sourceforge.net/apps/mediawiki/amos/index.php?title=AMOS>

- Ace files (including CAF)
  - Used by Phrap, CAP, other assemblers and viewers (e.g. consed, xgap)
  - Older, capillary-based. Focus more on read alignments and consensus, not on graph structure.
- More general formats, like GraphML, GraphViz dot

Richard Durbin <rd@sanger.ac.uk> 15 March, 2011