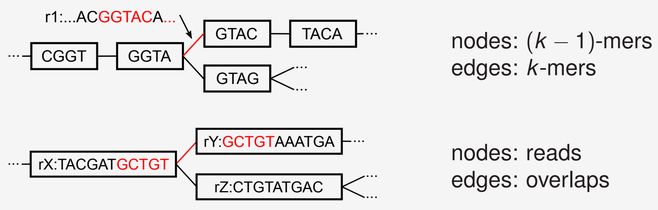
RGFA/BFA: convenient handling of compact assembly graphs

Giorgio Gonnella[1], and Stefan Kurtz[1]

[1] Research Group for Genome Informatics, Center for Bioinformatics, University of Hamburg, Bundesstraße 43, 20146 Hamburg, Germany

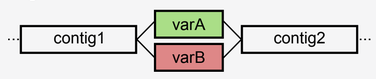
Assembly graphs

- sequence assemblers: reads → complete sequence
- often graph-based:



- usual output: contigs
- but: assembly graph is more informative
  - all possible solutions compatible with the data
  - allows understanding why assembly is interrupted
  - manually improve or finish the assembly (e.g. select paths based on additional info)

e.g. variants:



Towards a standard format

- Common problems:
  - assemblers use own graph representation (binary; edges as pointers in memory)
  - no output of graph or program-specific format
- FASTG [Jaffe et al., 2012]
  - extends FASTA (contigs output standard)
  - design problems, e.g. sequences are graph edges, complicates reverse complementing a single sequence
  - only adopted by very few tools
- GFA *Graphical Fragment Assembly* [Li, 2014]
  - text format, line-oriented, tab-separated
  - originally proposed by Heng Li, 2014
  - further collaborative development [Jackman et al., 2016]
  - until now: *limited scripting support*

GFA already adopted by:

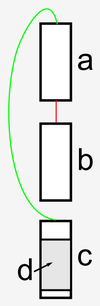
- assemblers (e.g. Abyss)
- alignment tools (Daliginer)
- variation analysis tools (vg)
- GUI editors (Bandage)
- format converters (gfatools)

GFA format

- Record types:
  - H header (metadata)
  - S segment (e.g. contig)
  - L link (suffix-prefix match)
  - C containment
  - P path (sequence of links)
- Fields (tab-separated):
  - mandatory (record type-specific)
  - optional (e.g. `VN:Z:1.0`)

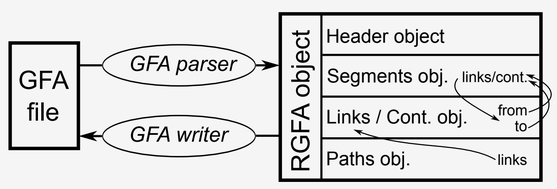
Example:

```
H VN:Z:1.0
S a CATGGCATGCTG
S b GCTGTTTCTA
S c GCCTGATGTGTT
S d CTGATGTG
L a + b + 4M
L c - a + 3M1D2M
C c + d + 2 8M
P p1 c-,a+,b+ *
```



RGFA library

- RGFA: (`http://github.com/ggonnella/rgfa`)
  - Ruby library: parsing, validating, editing GFA
  - free software (ISC license)
  - exploits GFA standard extension capability (RGFATools)
- Use cases:
  - implementation of own pipelines (reproducible / batch processing)
  - documentation of changes by interactive editors
  - rapid prototyping of graph editing algorithms



- Access to graph data:
  - header (as single line object)
  - iterate over segments/links/...;
  - segments/paths by name (hash table key)
  - links/cont. references in segments/paths
  - links of segment extremity (begin/end)

```
gfa.header.xy = "abcd"
gfa.segments; gfa.each_link do ...
gfa.segment(:sA)
gfa.segment(:sA).links[:from][:+]
gfa.links_to([:sB,:E])
```
(finds both: `L sA + sB -` and `L sB + sC +`)

- Interacting with GFA lines and fields:
  - methods: field names
  - new custom tags methods generated on the fly
  - field data converted from/to appropriate Ruby types

```
puts segment.sequence
segment.name = :newname
line.zz = "my new tag"
line.ab = 12 → ab:i:12
line.hh = {:a => 1, :b => [2,3]}
 → hh:J:"a":1,"b":[2,3]
```

BFA: a binary GFA counterpart

- BFA draft format:
  - represents same information as GFA
  - borrows representation conventions from BAM format
  - binary specification for each field
  - resulting file compressed using zlib
- e.g. large de Bruijn graph: GFA 68 MB; BFA 9 MB; Zip 11 MB
- Storage order: H/S/L/C/P
  - links/containments refer to segment ordinal number
  - paths refer to links ordinal number
- BFA ruby library: (`https://github.com/ggonnella/bfa`)
  - implements specification proposal
  - interconversion with GFA
  - BFA support for RGFA library

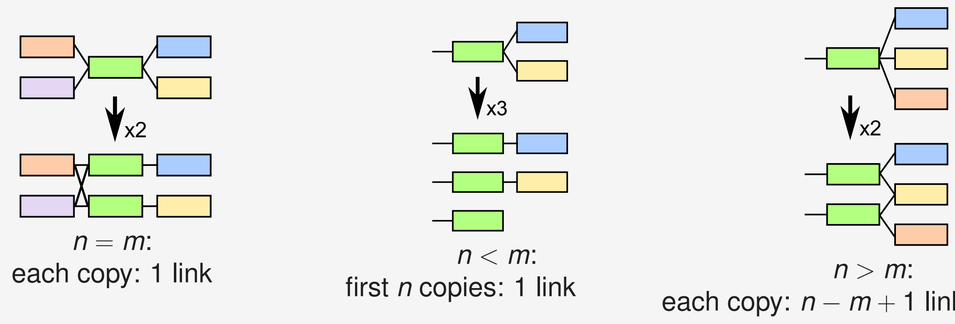
Representation of segments in BFA:

Field	Description	Type
n_segments	Number of segment records	`uint32_t (< MAXINT32T)`
Segments (length: n_segments)		
l_name	Length of segment name	`uint32_t`
name	Segment name	`char[l_name]`
l_seq	Uncompressed sequence length	`uint32_t`
seq	4-bit encoded read (see SAM spec)	`uint8_t[(l_seq+1)/2]`
n_optfields	Number of optional fields	`uint32_t`
Optional fields (length: n_header_tags times)		
tag	Two-character tag	`char[2]`
val_type	Value type	`char (AcCsSiIfZBJH)`
value	Tag value	`depends on val_type`

RGFA graph operations

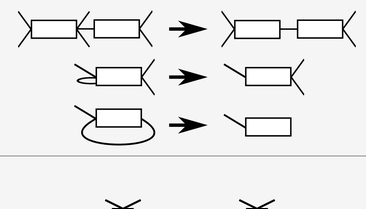
Multiplication

- replace segment *s* with copy number $m > 1$ by m identical segments
- smart assignment of n links to the copies (for one of *s* extremities):



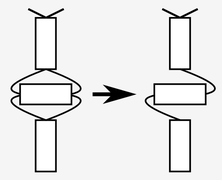
Removal of superfluous links

- not compatible with Hamilton path:
  - further links of extremity with *mandatory link*
  - self-links



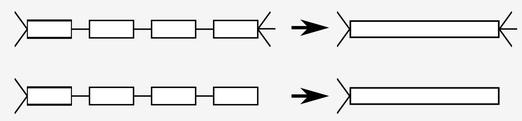
Random orientation

- segment has same links for both extremities; meaning: segment orientation is unknown
- idea: select random orientation and annotate



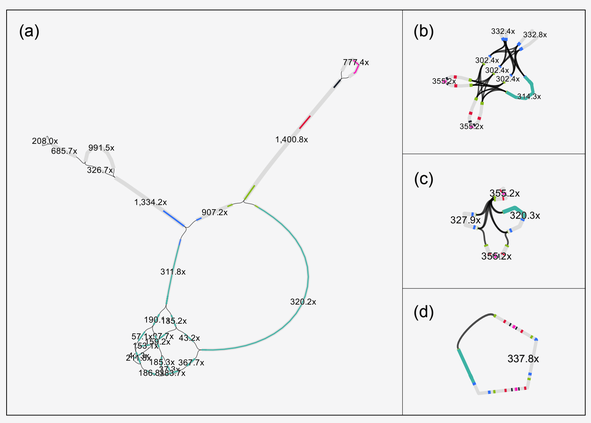
Linear paths merging

- Linear paths:
  - start/end: segment extremity with 1 link
  - internal: both extremities 1 link
- Merge operation:
  - collapse into single segment
  - merge sequence / update links



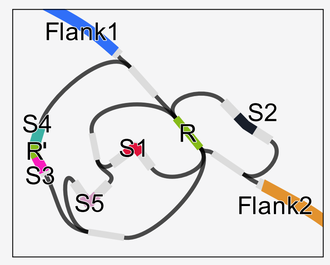
Case studies

(1) **Finish repetitive sequence assembly**
e.g. fosmid insert containing repeats; (a) result of SPAdes; (b,c,d) different stages of simplification using multiplication, removal of superfluous links, random orientation and merging of linear paths



(2) **Develop a graph algorithm**

- CRISPRs: Clustered Regularly Interspaced Short Palindromic Repeats
  - common feature of prokaryotic genomes
  - conserved directed repeat (24–47 bp)
  - unique flanks and spacer (26–72 bp)
- CRISPR signature in a de Bruijn graph:
  - Flank1/Flank2: unique flank sequences
  - R: exact repeat; R': approx repeat instance
  - S1–S5: spacer sequences



(3) **Compare GFA files** (GFAdiff)

File *gfa1*:

```
H VN:Z:1.0 aa:i:98
S a CAGGGCATGCTG
S b GCTGTTTCTA
S c CCTGATGTGTT
S d CTGATGTG
L a + b + 4M
L a - c + 2M1D2M
C c + d + 1 8M
```

File *gfa2*:

```
H VN:Z:1.0 aa:i:99
S a CAGGGCATGCTG
S b GCTGTTTCTA xx:Z:a
S c CCTGATGTGTT
L a + b + 4M
L c - a + 2M1I2M
```

Report:

```
< [H/tag/diff] aa:i:98
> [H/tag/diff] aa:i:99
> [S/tag/exclusive/a] xx:Z:text
< [S/exclusive] S d CTGATGTG
< [C/exclusive-S] C c + d + 1 8M
```

RGFA script:

```
g=RGFA.from_file("gfa1")
g.header.aa = 99
g.segment(:b).xx = "a"
g << "S d CTGATGTG"
g << "C c + d + 1 8M"
```

Conclusions

RGFA (`https://github.com/ggonnella/rgfa`) [Gonnella and Kurtz, 2016] is a powerful scripting library for the GFA format. Example applications include implementing reproducible finishing of an assembly, and documenting by GFAdiff manual changes introduced using interactive GUI editors.

BFA (`https://github.com/ggonnella/bfa`) is a proposal for encoding GFA information in a compact binary format. Our Ruby library allows interconversion with the GFA format and access to BFA data using RGFA.

Future work on RGFA will involve implementing time-critical steps in C. Furthermore, we plan to start a collaborative development effort for the further development of a binary companion format to GFA on the basis of our proposed BFA format.

References

Gonnella, G. and Kurtz, S. (2016).
RGFA: powerful and convenient handling of assembly graphs.
Peer J. Preprints, page 4:e2381v1.

Jackman, S. et al. (2016).
Graphical Fragment Assembly (GFA) Format Specification.
`https://github.com/pmelsted/GFA-spec`.
(last accessed 2016-06-06).

Jaffe, D. B., MacCallum, I., Rokhsar, D. S., and Schatz, M. C. (2012).
The FASTG Format Specification (v1.00).
`http://fastg.sourceforge.net/FASTG_Spec_v1.00.pdf`.
(last accessed 2016-06-06).

Li, H. (2014).
A proposal of the graphical fragment assembly format.
`http://lh3.github.io/2014/07/19/a-proposal-of-the\-grapical-fragment-assembly-format/`.
(last accessed 2016-06-06).

Mail: `gonnella@zbh.uni-hamburg.de` WWW: `http://www.zbh.uni-hamburg.de/gonnella`