1 The GfaPy interface to custom record types

The GFA 2 format can be extended by defining new record types. According to the specification a core implementation of the format can ignore these non-standard lines. Custom records can be used for extending the range of possible applications of the format.

In GfaPy, the data contained in a GFA file is represented by instances of the Gfa class. The list of lines of a given type are accessed as properties of the instance, named after the line type, e.g. segments and custom_records. The method custom_records_of_type(t) allows to find all custom records where t is the content of the first field (record_type) in those records.

Each line in the GFA file consists of TAB-separated fields. The first field is a string defining the record type. It is followed by a number of positional fields (except for header lines, which have none), i.e. fields whose datatype and field name is implicit from their position in the line. Furthermore, each line can be complemented by tags, optional fields where the name and datatype is written explicitly.

In GfaPy a GFA line is represented by an object of a subclass of gfapy.Line. The data in positional fields of standard record types is accessed using properties, which are named according to the field name (e.g. sequence). GfaPy knows the name of each field and how to handle the data contained in it, as this is defined by the GFA specification. So GfaPy validates all strings contained in the GFA file. After validating the string representation in a field GfaPy instantiates Python objects of appropriate classes to represent the data (e.g. integers). If the user edits the data, GfaPy checks if the content remains valid (e.g. for an
integer $i$ defining a position it verifies $i \leq 0$). Furthermore, fields referencing other records (such as segment names in edge records) are replaced by references to the corresponding instances. For example, the from field of a link line contains the name of a segment line and is replaced by a reference to the segment instance. For each reference, lists of backreferences are constructed, so that a user can e.g. find all links referencing a given segment via the links property of the segment instance.

For custom records, by default, GfaPy does not have the information for providing the same convenient interface as for the standard GFA record types. As the field names are unknown, the field content is accessed using generic names which only reflect the position of the field in the line (field1, field2, ...) but not its semantic. As the datatype is unknown, the content is not validated and stored as Python strings. References to other lines are left unresolved.

However, the user may define an extension, which describes the contents of the records. This allows GfaPy to provide the same interface and functionality available for standard GFA record types, as described above.

1.1 An example of custom record types

To exemplify how GfaPy handles custom records, we here consider custom record types for representing taxon-specific subgraphs of assembly graphs obtained from metagenomic read-sets.

Assembling metagenomic reads leads to assemblies with mixtures of sequences originating from the different organisms contained in the metagenomic sample. Specialized assemblers, such as Ray Meta (Boisvert et al., 2012), MEGAHIT (Li et al., 2015) and metaSPAdes (Nurk et al., 2017), or post-assembly binning tools, such as CONCOCT (Alneberg et al., 2014) can be used for assigning these sequences to species.

Often the assignment of a contig to a species is ambiguous, due to lack of data. Moreover, some sequences are actually shared between different species. If the result of an assembly is represented in form of an assembly graph, it would be useful to store the taxonomic assignments in the graph. As the GFA specification does not provide records specialized for this kind of information, we designed corresponding custom records. In particular, we make use of two custom record types, namely T-records (see Table 1) for taxa and M-records (see Table 2) for assignments of a segment to a taxon.

As these custom records contain specialized datatypes and references to other records, it is useful to write a GfaPy extension to validate and process them. The code for the GfaPy extension is explained in the following text. It is available in the tests directory of the GfaPy code repository at https://github.com/ggonnella/gfapy.

The content of a sample GFA 2 file with these custom record types is given in Figure 1.
<table>
<thead>
<tr>
<th>Field</th>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tid</td>
<td>identifier_gfa2</td>
<td>Taxon ID</td>
</tr>
<tr>
<td>UL (tag)</td>
<td>string</td>
<td>Link to an external resource describing the taxon</td>
</tr>
</tbody>
</table>

Table 1: Definition of the fields in the custom T-records.

<table>
<thead>
<tr>
<th>Field</th>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mid</td>
<td>optional_identifier_gfa2</td>
<td>ID of the metagenomic assignment record (or *).</td>
</tr>
<tr>
<td>tid</td>
<td>identifier_gfa2</td>
<td>Taxon ID, reference to a T-record</td>
</tr>
<tr>
<td>sid</td>
<td>identifier_gfa2</td>
<td>Segment ID, reference to an S-record</td>
</tr>
<tr>
<td>SC (tag)</td>
<td>integer</td>
<td>Error probability of the assignment expressed as Phred-style score</td>
</tr>
</tbody>
</table>

Table 2: Definition of the fields in the custom M-records.

S sA 1000 *
S sB 1000 *
M assignment1 t123 sA SC:i:40
M assignment2 t123 sB
M * B12c sB SC:i:20
T B12c
T t123 UL:Z:http://www.taxon123.com

Figure 1: Content of a GFA 2 formatted file with custom record types T and M.
<table>
<thead>
<tr>
<th>RECORD_TYPE</th>
<th>record type descriptor (M or T in the example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSFIELDS</td>
<td>ordered dictionary specifying the datatype (see Table 4) for each positional field; the fields must be inserted in the dictionary in the order they are found in the line (default: empty dictionary)</td>
</tr>
<tr>
<td>TAGS_DATATYPE</td>
<td>dictionary specifying the datatype of predefined (i.e. upper case) optional tags (default: empty dictionary)</td>
</tr>
<tr>
<td>NAME_FIELD</td>
<td>name of the field which contains the name of the record; this is required for extension records which can be referred to from other lines (default: None)</td>
</tr>
</tbody>
</table>

Table 3: Constants for the definitions of subclasses of gfapy.Line.

```python
from collections import OrderedDict

class Taxon(gfapy.Line):
    RECORD_TYPE = "T"
    POSFIELDS = OrderedDict({"tid": "identifier_gfa2"})
    TAGS_DATATYPE = {"UL": "Z"}
    NAME_FIELD = "tid"

Taxon.register_extension()
```

Figure 2: Code for the class for T-records.

### 1.2 Creating custom gfapy.Line subclasses

In GfaPy a new record type is defined by creating custom subclasses of the class gfapy.Line. Such subclasses can easily be introduced, as they only require the definition of some constants (see Table 3) and calling the method register_extension(). M-records have references to segments and to T-records. To correctly handle these references, the register_extension method can optionally be supplied with a list of triples (fieldname, classname, backreferences) as parameter, where fieldname is the name of the field in the corresponding record containing the reference, classname is the name of the class, the reference refers to, and backreferences is the name of a property in instances of the class classname, which will containing backreferences to the lines of type defined in the extension. For example for M-records, the list contains two triples, as two fields (sid and tid) refer to other lines (segments and T-records, respectively). The backreferences in both lines are named metagenomic_assignments, i.e. the list of the M-records which refer to a segment instance s can be retrieved using the property s.metagenomic_assignments. The code for subclasses for the T-records and M-records is shown in Figures 2 and 3, respectively.
from collections import OrderedDict

class MetagenomicAssignment(gfapy.Line):
    RECORD_TYPE = "M"
    POSFIELDS = OrderedDict([("mid","optional_identifier_gfa2"),
                               ("tid","identifier_gfa2"),
                               ("sid","identifier_gfa2")])
    TAGS_DATATYPE = {"SC":"i"}
    NAME_FIELD = "mid"

    MetagenomicAssignment.register_extension(references=
                                          ["sid", gfapy.line.segment.GFA2, "metagenomic_assignments"],
                                          ["tid", Taxon, "metagenomic_assignments"])

    Figure 3: Code for the definition of the M-records.

1.3 Defining custom datatypes

In GfaPy the datatypes predefined in the GFA specification are implemented by classes providing functions for decoding, encoding and validating the corresponding data. The predefined datatypes for fields in GFA 2 are listed in Table 4.

For custom records, one sometimes needs datatypes not yet available in the GFA specification. For example, it would be useful to have a datatype for a taxon identifier of the T- and M-records. If one chooses the taxon identifier to be either of the form taxon:<n> (where <n> is a positive integer) or consist of letters, numbers and underscores only, then in GfaPy one could define the corresponding datatype taxon_id as shown in Figure 4.

2 Differences between RGFA and GFApy

Naturally, GfaPy and the Ruby library RGFA (Gonnella and Kurtz, 2016) have much in common. In fact, for the initial development of GfaPy we used the RGFA code as a template. However, the recoding in Python was not trivial, as RGFA uses a lot of Ruby specific language features (such as metaprograming, blocks and the extension of standard library classes), which are not available or expressed in a different way in Python. Thus even when restricting to the subset of features of GfaPy already available in RGFA, the code has many differences (except for the language, of course).

More importantly, GfaPy provides many new features compared to RGFA, such as:

- support of both GFA 1 and GFA 2,
- the interconversion between those versions,
- the availability of an interface for writing code that is independent from the version of the GFA-format,
- support of custom line types of GFA 2.
import re

class TaxonID:
    def validate_encoded(string):
        if not re.match(r"taxon:(\d+)$", string) and \
            not re.match(r"^[a-zA-Z0-9_]+$", string):
            raise gfapy.ValueError("Invalid taxon ID: {}".format(string))
    def decode(string):
        TaxonID.validate_encoded(string)
        return string
    def validate_decoded(obj):
        if isinstance(obj, Taxon):
            TaxonID.validate_encoded(obj.name)
        else:
            raise gfapy.TypeError("Invalid type for taxon ID: "+"{}".format(repr(obj)))
    def encode(obj):
        TaxonID.validate_decoded(obj)
        return obj

# register the new datatype
gfapy.Field.register_datatype("taxon_id", TaxonID)

# use the datatype in Taxon and MetagenomicAssignment
Taxon.DATATYPE["tid"] = "taxon_id"
MetagenomicAssignment.DATATYPE["tid"] = "taxon_id"

Figure 4: Definition of a custom datatype taxon_id.
In the following, we explain in detail the modifications relative to RGFA and the additional implementations in GfaPy.

### 2.1 Field datatypes

RGFA contains three modules (FieldParser, FieldValidator and FieldWriter) with code for parsing, validating and encoding the field data. Validation is limited to the encoded data, and is based on regular expressions. The handling of the different kind of fields are directly coded in the modules methods.

As GfaPy handles both GFA 1 and GFA 2, the number of datatypes to handle fields almost doubled from 15 to 28. To keep the code manageable, it was reorganized in such a way that each datatype has its own module with specific methods for parsing, validating and encoding it. The modules for the 28 predefined datatypes are in the field directory. Besides providing a better code organization, an important advantage of the modular system is that a user can specify custom datatypes by adding a corresponding module to the code, thus considerably improving the extensibility of GfaPy. Furthermore, validation is no longer limited to a check of whether a given regular expression matches. Instead, for validation one can specify any function, which raises an exception if invalid data is provided. For example, the validity of data stored in JSON (J) tags is checked by trying to parse the JSON string using the python json library. Finally, validation can be extended to decoded fields. For example, the decoded content of a numeric array (B) tag can be directly validated by checking if all the array elements are integers or floats, instead of converting each element into its string representation and checking if the regular expression provided in the GFA specification matches.
2.2 Record type description

In RGFA, record types are defined via a system of constants. In GfaPy this system was modified in such a way that constants allow to specify that, for example, a particular field contains the identifier for a line, or to describe which fields contain references to other records. The new system of constants in GfaPy allows the user to more easily create extensions for handling custom records, as described in Section 1 of this document.

2.3 References to records in the graph object

In RGFA, records are represented by Line instances stored in different variables (e.g. @segments) of an RGFA object. Depending on the record type, a hash table or array is used to store the references.

GFA 2 introduces new record types, namely edges, ordered groups, unordered groups and fragments. Furthermore, GfaPy handles both GFA 1 and GFA 2, as well as custom record types. To enable this, the interface to the records storage was generalized. A single instance variable (_records) of the GFA object contains a collection of dictionaries. Appropriate keys are automatically selected, depending on the record type and the presence or absence of a record identifier. Generic code can now handle the different record types (e.g. collecting all records of a given type, such as all segments). Furthermore, there is a common interface to methods accessing the records of the different GFA versions. For example, the dovetails method of the GFA class returns a list of all dovetail overlaps. While these are specified by a specific form of E-lines in GFA 2, GFA 1 uses L-lines. GfaPy abstracts from the differences and allows the user to write code which works for GFA 1 and GFA 2.

2.4 References between pairs of records

Inter-record references are present in GFA 1 and GFA 2. In GFA 1, all references (in Link-, Containments- and Paths-lines) are to segment lines. In GFA 2, edges, gaps and fragments reference segment lines only, but paths and sets may also reference other line types, including other paths and sets.

To handle the more complex reference system of GFA 2, GfaPy includes a generic interface for references. This interface is used, for example, to enforce inter-line dependencies, so that the deletion of a record is automatically cascaded to the records which depend on it: e.g. removing a segment leads to the removal of all records which refer to it and of all records dependent on the latter, etc. Finally, the flexibility of the new references system in GfaPy allows the user to easily incorporate inter-line references in custom record extensions.
2.5 Links for graph traversal

In RGFA the user must always access the graph object (instance of the RGFA class), in order to traverse the graph. For example, to retrieve the list of links of a segment, the links_of method of RGFA is provided.

In GfaPy, the system of inter-record references described above allows to directly traverse the graph, using a simple and intuitive interface. For example, the list of dovetails overlaps of a segment is contained in the dovetails property of the segment.

2.6 Interconversion and version-independent interface

GFA 2 introduces representations for graph elements which are not available in GFA 1 (fragments, gaps, internal alignments, sets). All other graph elements are available in GFA 1 and GFA 2, but with differences (according to the GFA specifications), that cannot be neglected. The only common record type that has not changed in GFA 2 (compared to GFA 1) is the header. Segments are very similar in GFA 1 and GFA 2, with an additional field (sequence length) and less restrictions for the sequence alphabet in GFA 2. Edge lines have fundamentally been modified in GFA 2: the two lines types L and C in GFA 1 were unified and the alignment positions must now be specified for all edges. Some E-lines in GFA 2 cannot be converted to GFA 1, namely internal alignments. GFA 2 paths have a completely different syntax compared to GFA 1 and may include references to edges, segments, and other paths.

For edges, segments and paths, GfaPy provides methods for computing the equivalent representation of the records in the two GFA versions (if possible). These methods not only allow for format interconversion, but also enable the user to access these graph elements with a generic interface, which is independent on the GFA version.

2.7 Trace alignments

GFA 2 allows to specify alignments as traces. GfaPy represents traces using the class Trace. As the same field can contain, in different lines, different alignment representations (CIGAR strings, traces, and undefined alignments), in GfaPy a generic class Alignment with a common interface has been introduced. It automatically recognizes the kind of alignment representation and creates objects of the appropriate class.

2.8 Last positions

In GFA 2, the begin and end coordinates of alignments are given in E- and F-lines. The specification requires that the suffix $ is added to a coordinate, if and only if the coordinate is equal to the length of the segment.

To represent coordinates with suffix $, GfaPy introduces a new class LastPos. The class constructor allows to parse a string containing a coordinate. It returns an instance of class
int or LastPos depending on the presence or absence of the suffix $. Further methods, such as simple arithmetic operations, allow for generic handling of the values of position fields. For example, subtracting a non-zero positive value from a LastPos instance correctly returns an int.

2.9 Placeholders

Compared to GFA 1, there are more fields in GFA 2 that may contain an unspecified value, signified by the symbol *. While in RGFA these values are handled by the code for a field, GfaPy generalizes the handling of the unspecified values by introducing the class Placeholder. This makes the code more flexible.

2.10 Fragments

The class Fragment of GfaPy implements fragment records in GFA 2. Fragment records represent relations of segments and external sequences, such as read to contig alignments. Methods are provided for retrieving all external sequences to which a segment is related, as well as all segments to which an external sequence is related.

2.11 Ordered and unordered groups

GFA 2 uses a different syntax for storing graph paths (called ordered group), and introduces a new kind of group named set or unordered group. GfaPy introduces the classes OrderedGroup and UnorderedGroup to handle these record types. Additional complexity accrues by the fact that the user may imply the presence of some graph elements, without including them in the specification of the group. For example, a path containing two segments includes the edge between them, even if not included in the path items list. The two classes for groups provide methods computing the captured segments and edges lists for paths and sets. Also, paths are only valid if the specified elements are contiguous, which is checked by the methods of the class OrderedGroup. Finally, GfaPy provides a common interface for paths in GFA 1 and GFA 2 allowing the user to write generic code for both versions of GFA.

2.12 Gaps

GfaPy introduces the class Gap for handling the gap record type, which was introduced in GFA 2. The interface is similar to that for edge lines. RGFA does not support gap records, as these are not available in GFA 1.
2.13 Custom records

The class CustomRecord of GfaPy handles GFA 2 records with a non-standard record type. Methods are provided to access the custom record fields and tags, which are automatically recognized using a heuristic. Furthermore, GfaPy defines the simple extension definition system described in Section 1 of this document.

RGFA does not support custom records, as these are not available in GFA 1.

2.14 Object creation and conversion

In RGFA objects are created using the constructor (class method new) or by conversion from other objects (e.g. a string representation), using .to_<CLASS> methods (for example string.to_rgfa_line). In GfaPy the constructor is used (e.g. Line(string)) for the creation as well as for the conversion from other objects.

3 Evaluation of the performance of GfaPy

While the implementation of GfaPy in a scripting language ensures the required flexibility in its user interface, this choice limits the performance of the library for large files. In this section, we describe a set of benchmark experiments to investigate which resources are necessary in GfaPy for handling GFA files of different size.

3.1 Methods

We generated sets of GFA files using the script gfapy-randomgraph. This Python script generates GFA files with a specified number of segments (consisting of random sequences) and a specified number of dovetail overlaps per segment.

We created a first set of files with a varying number of segments, ranging from 1 000 to 2 048 000, and 2 dovetail overlaps per segment on average (dataset 1). To investigate the impact of the degree of connectivity of the graph on the performance of GfaPy, we created a second set of files with a constant number of segments (4 000) and a varying number of dovetail overlaps per segment, ranging from 2 to 256 (dataset 2).

In all files, each segment consists of an explicit sequence (i.e. not a sequence placeholder) with a fixed length of 100 bp.

To each of the generated files, we applied the GfaPy-based scripts gfapy-convert (interconversion between GFA 1 and GFA 2) and gfapy-mergelinears (merging of linear path).

The memory peak and running time were collected using the shell script gfapy-benchmark-collectdata. This measures the wall clock running time of the
process using the GNU `time` utility and collects the memory peak data of the running process, as reported by the Linux operating system in the `proc` virtual filesystem. All runs were performed with Python version 3.3, a single thread on an Intel Xeon E5-2680 CPU running at 2.70 GHz. For each experiment we report the average running time and space requirement over three runs.

The script `gfapy-plot-benchmarkdata.R` was used to plot the data using the R library `ggplot2` and fit the parameters of a linear regression models using the R function `nms`.

All the mentioned scripts are available in the `benchmarks` directory of the `GfaPy` github repository (https://github.com/ggonnella/gfapy).

### 3.2 Results

The conversion of GFA 2 files to GFA 1-format required running time and space requirement linear in the number of input lines, see Figure 5 and Figure 6. In particular, for dataset 1 (dataset 2), `GfaPy` required about 0.8 s (0.9 s) and 2.0 Mb (2.3 Mb) for 1 000 lines of input.

The merging of linear paths is a more complex operation, which requires to traverse the entire graph, delete segments and edges and add new ones. For both sets of files, the running time and space requirement of linear path merging was linear in the number of input lines, see Figure 7 and Figure 8. For both datasets `GfaPy` required 0.6 s and at most 2.1 Mb of memory for 1 000 lines of input.
Figure 6: Running time (left) and space requirement (right) for the conversion of a GFA 2 file into a GFA 1 file, containing a fixed number of segments (4000) and a varying average number of dovetail overlaps per segment indicated on the X-axis.

Figure 7: Running time (left) and space requirement (right) for merging of linear paths for data specified in a GFA 2 file. One third of the lines were segments and the average number of edges per segment was 2.
Figure 8: Running time (left) and space requirement (right) for merging of linear paths for data specified in a GFA 2 file. The files contained 4000 segments and a varying average number of dovetail overlaps per segment indicated on the X-axis.

References


