graf-python Documentation

Release 0.3.0

Peter Bouda

September 15, 2014

Contents

The library graf-python is an open source Python implemenation to parse and write GrAF/XML files as described in ISO 24612.

The project's homepage is: http://media.cidles.eu/poio/graf-python/

Contents

CHAPTER 1

Introduction to graf-python

The library graf-python is a Python implemenation to parse files GrAF/XML files as described in ISO 24612. The parser of the library create an annotation graph from the files. The user may then query the annotation graph via the API of graf-python. This documentation gives some examples how to acccess data and how to transform it for further processing in linguistic and computational libraries like networkX, numpy and NLTK.

References:

- GrAF Wiki (http://www.americannationalcorpus.org/graf-wiki)
- ISO 24612 (http://www.iso.org/iso/catalogue_detail.htm?csnumber=37326)

1.1 How to use the library

1.1.1 Parsing GrAF

The first step is to initialize the parser:

```
   import graf
   gparser = graf.GraphParser()

You can then directly pass the filename of the GrAF header to the parser:

graph = gparser.parse("filename.hdr")
```

The parser then collects all the dependencies from this header. You might also pass the file name of any GrAF/XML file to the parser. The parser then loads all files that are dependencies of that file as described in its header.

Alternativily the parser also accepts and open file stream. Now we have a GrAF object and it is possible to verify the nodes, regions, edges and their respectives annotations.

```
# Checking the nodes
for node in graph.nodes():
    print(node)

# Checking the regions
for region in graph.regions():
    print(region)

# Checking the edges
for edge in graph.edges():
    print(edge)
```

1.1.2 Querying GrAF graphs

For a real-world example how to use the GrAF API in Python we will use data from the project "Quantitative Historical Linguistics". The project publishes data as GrAF/XML files that are ready to use with the parser. Here we will use the XML files of the dictionary "Thiesen, Wesley & Thiesen, Eva. 1998. Diccionario Bora—Castellano Castellano—Bora", which are available as a ZIP package:

http://www.quanthistling.info/data/downloads/xml/thiesen1998.zip

Download and extract the files to a local folder. The following example code will extract all head and translation annotations from the XML files and write them into a separate, tab-separated text file.

First, create a parser object and parse the file "dict-thiesen1998-25-339-dictinterpretation.xml" that you extracted from the ZIP file:

```
import graf

gparser = graf.GraphParser()
g = parser.parse("dict-thiesen1998-25-339-dictinterpretation.xml")
```

This will parse the file and all its dependencies into a GrAF object that we can query now. In this case the only dependency is the file "dict-thiesen1998-25-339-entries.xml" that contains regions of dictionary entries that link to the basic data file, and annotations for each of those dictionary entries. We will use the entry nodes to find each "head" and "translation" annotation that are linked to the entry nodes via edges in the graph.

Next, open the output file:

```
f = codecs.open("heads_with_translations_thiesen1998.txt", "w", "utf-8")
```

Then you may loop through all the nodes in the graph. For each node that has a label ending in "entry" we will follow all the edges. The edges that have label "head" or "translation" link to the annotations nodes we want to extract:

```
# loop through all nodes in the graph
for (node_id, node) in g.nodes.items():
    heads = []
    translations = []

    # if the node is a dictionary entry...
    if node_id.endswith("entry"):

        # loop thropugh all edges that are connected
        # to the entry
        for e in node.out_edges:
            # if the edge has a label "head"...
            if e.annotations.get_first().label == "head":
                # get the "head" annotation string
                heads.append(e.to_node.annotations.get_first().features.get_value("substring"))

            # if the edge has a label "translation"...
            elif e.annotations.get_first().label == "translation":
                # get the "translation" annotation string
                translations.append(e.to_node.annotations.get_first().features.get_value("substring"))
        # write all combinations of heads and translations
        # to the output file
        for h in heads:
            for t in translations:
                f.write(u"{0}\t{1}\n".format(h, t))
```

This will write heads and translations to the file, separated by a tab. Don't forget to close the file in the end:

```
f.close()
```

The complete script is available in the Github repository of graf-python:

https://github.com/cidles/graf-python/blob/master/examples/query_quanthistling_graf.py

1.1.3 Translation Graphs from GrAF/XML files

In this tutorial we will demonstrate how to extract a translation graph from data in digitized dictionaries. The translation graph connects entries in dictionaries, via annotation for "heads" and "translations" within the dictionary. We will demonstrate how to visualize this data with a plotting library and how to export parts of the graph to JSON for interactive visualizations in the web.

You can download this tutorial as IPython notebook here:

https://github.com/cidles/graf-python/blob/master/examples/Translation%20Graph%20from%20GrAF.ipynb

Or as a Python script here:

https://github.com/cidles/graf-python/blob/master/examples/Translation%20Graph%20from%20GrAF.py

Data

For this tutorial we will use data from the project "Quantitative Historical Linguistics". The website of the project provides a ZIP package of GrAF/XML files for the printed sources that were digitized within the project:

http://www.quanthistling.info/data/downloads/xml/data.zip

The ZIP package contains several files encoded as described in the ISO standard 24612 "Linguistic annotation framework (LAF)". The QuantHistLing data contains dictionary and wordlist sources. Those were first tokenized into entries, for each entry you will find annotations for at least the head word(s) ("head" annotation) and translation(s) ("translation" annotation) in the case of dictionaries. We will only use the dictionaries of the "Witotoan" component in this tutorial. The ZIP package also contains a CSV file "sources.csv" that contains some information for each source, for example the languages as ISO codes, type of source, etc. Be aware that the ZIP package contains a filtered version of the sources: only entries that contain a Spanish stem that is included in the Spanish swadesh list are included in the download package.

For a simple example how to parse one of the source please see here:

http://graf-python.readthedocs.org/en/latest/Querying%20GrAF%20graphs.html

What are translation graphs?

In our case, translation graphs are graphs that connect all spanish translation with every head word that we find for each translation in our sources. The idea is that spanish is some kind of interlingua in our case: if a string of a spanish translation in one source matches a string in another source this will only be ''one'' node in our graph. For the head words, this is not the case: matching strings in head words in different source are different nodes in the graph. This holds even if the different sources describe the same language, as different sources will use different orthographies.

To fullfil that need, head words are internally represented as a string with two parts: the head word and its source. Both parts are seperated by a pipe symbol "|". For example, in a DOT file such a node looks like this:

> "ócájil|thiesen1998" [lang=boa, source=thiesen1998_25_339];

The square brackets contain additional attributes here. These attributes are not part of the node's name, they contain just additonal information the user wants to store with the nodes.

In comparison, a spanish translation looks like this:

"vaca" [lang=spa];

There is no attribute "source" here, as this translation might occur in several sources. An edge connecting the two nodes looks like this:

"vaca" – "ócájil|thiesen1998";

To handle such graphs our scripts use the NetworkX Python library. It is kind of a standard in scientific graph computing with Python (remark: I started with the pygraph library, which has more or less the same API but by far not enough operations to compute with graphs later).

Requirements

The following Python libraries are required to process the GrAF/XML files and create the translation graphs:

- NetworkX: http://networkx.lanl.gov/
- graf-python: https://github.com/cidles/graf-python
- NLTK: http://nltk.org

To visualize the graphs you have to install matplotlib:

- matplotlib: http://matplotlib.org/

When you have installed all the libraries you are able to import the following modules:

In[22]:

```
import os
import csv
import codecs
import re
import glob

import networkx
import graf
import nltk
```

Get Witotoan sources

Change to the directory where you extracted the ZIP archive that you downloaded from the QuantHistLing website:

In[2]:

```
os.chdir("h:/Corpora/qlc/graf")
```

Now we open the file "sources.csv" and read out all the sources that are part of the component "Witotoan" and that are dictionaries. We will store a list of those source in `witotoan_sources`:

In[3]:

```
sources = csv.reader(open("sources.csv", "rU"), delimiter="\t")
witotoan_sources = list()
for source in sources:
    if source[5] == "Witotoan" and source[1] == "dictionary":
        witotoan_sources.append(source[0])
```

GrAF to NetworkX

Next we define a helper function that transform a GrAF graph into a networkx graph. For this we traverse the graph by querying for all entries. For each entry we look for connected nodes that have "head" or "translation" annotation. All of those nodes that are Spanish are stored in the list `spa`. All non-Spanish annotations are stored in `others`. In the end the collected annotation are added to the new networkx graph, and each spanish node is connected to all the other nodes for each entry:

In[51]:

```
def graf_to_networkx(graf, source = None):
    g = networkx.Graph()
    for (node_id, node) in graf.nodes.items():
        spa = list()
        others = dict()
        if node_id.endswith("entry"):
            for e in node.out_edges:
                if e.annotations.get_first().label == "head" or e.annotations.get_first().label == "t
                    # get lang
                    for n in e.to_node.links[0][0].nodes:
                        if n.annotations.get_first().label == "iso-639-3":
                            if n.annotations.get_first().features.get_value("substring") == "spa":
                                spa.append(e.to_node.annotations.get_first().features.get_value("subs
                                break
                            else:
                                others[e.to_node.annotations.get_first().features.get_value("substri
                                break
        if len(spa) > 0:
            for head in spa:
                g.add_node(head, attr_dict={ "lang": "spa" })
                for translation in others:
                    g.add_node(u"{0}|{1}".format(translation, source), attr_dict={ "lang": others[tra
                    g.add_edge(head, u"{0}|{1}".format(translation, source))
    return g
```

Parse GrAF/XML files

Now we parse all the XML files of the extracted ZIP package. For this we traverse through all the directories that have a name in 'witotoan_sources'. The files we are looking for are the "-dictinterpretation.xml" files within each directory, as those contain the annotations for "heads" and "translations".

First we create an empty list `graphs` that will later store all the networkx graphs:

In[52]:

```
parser = graf.GraphParser()
graphs = []
```

Then we loop through all the Witotoan sources, parse the XML files and transform the graphs into networkx graph by calling the helper function that we defined above. We print a progress report within the loop, as parsing and transformation might take some time:

In[53]:

```
for d in witotoan_sources:
    for f in glob.glob(os.path.join(d, "dict-*-dictinterpretation.xml")):
        print("Parsing {0}...".format(f))
        graf_graph = parser.parse(f)
```

```
        g = graf_to_networkx(graf_graph, d)
        graphs.append(g)
print("OK")
```

```
Parsing thiesen1998dict-thiesen1998-25-339-dictinterpretation.xml...
Parsing minor1987dict-minor1987-1-126-dictinterpretation.xml...
Parsing minor1971dict-minor1971-3-74-dictinterpretation.xml...
Parsing burtch1983dict-burtch1983-19-262-dictinterpretation.xml...
Parsing leach1969dict-leach1969-67-161-dictinterpretation.xml...
Parsing walton1997dict-walton1997-9-120-dictinterpretation.xml...
Parsing preuss1994dict-preuss1994-797-912-dictinterpretation.xml...
Parsing rivet1953dict-rivet1953-336-377-dictinterpretation.xml...
Parsing griffiths2001dict-griffiths2001-79-199-dictinterpretation.xml...
OK
```

Merge all graphs

Now we can merge all the individual graphs for each source into one big graph. This will collapse all Spanish nodes and so connect the nodes that have a common Spanish translation:

In[54]:

```
import copy
combined_graph = copy.deepcopy(graphs[0])
for gr in graphs[1:]:
    for node in gr:
        combined_graph.add_node(node, gr.node[node])
    for n1, n2 in gr.edges_iter():
        combined_graph.add_edge(n1, n2, gr.edge[n1][n2])
```

We count the nodes in the graph and the number of connected components to get an impression how the graph "looks". The number of nodes is much higher than the number of connected components, so we already have a lot of the nodes connected in groups, either as a consequence from being part of one dictionary entry or through the merge we did via the Spanish node:

In[55]:

```
len(combined_graph.nodes())
```

Out[55]:

```
73022
```

In[56]:

```
networkx.algorithms.components.number_connected_components(combined_graph)
```

Out[56]:

```
17021
```

Connect nodes with the same stem

The next step is to connect spanish translations that contain the same stem. For this we first remove certain stop words from the translation (list of stopwords from NLTK). There are two cases then: just one word remains, or more than one word remains.

We have two options now what to do with the latter: either they are not connected with anything at all (default behaviour), or each word is stemmed and the translation is connected with every other translation that contain the same stems. Right now this results in many connections that look not very useful. This should be done in a more intelligent way in the future (for example find heads of phrases in multiword expressions and only connect those; split the weight of the connections between all stems and work with weighted graphs from this step on; ...).

To connect the spanish translations the script adds additional "stem nodes" to the graph. The name of these nodes consists of a spanish word stem plus a pipe symbol plus the string "stem". These nodes look like this in a dot file:

"tom|stem" [is_stem=True];

The introduction of these nodes later facilites the output of translation matrixes, as you can just search for stems within the graph and only output direct neighbours with spanish translations. It would also be possible to directly connect the spanish translations if they have a matching stem, but then the graph traversal to find matching translations and their heads is a bit more complex later.

First we create a stemmer object from the SpanishStemmer in NLTK:

In[57]:

```
from nltk.stem.snowball import SpanishStemmer
stemmer = SpanishStemmer(True)
```

We create the list of stopwords and encode them as unicode strings:

In[58]:

```
combined_graph_stemmed = copy.deepcopy(combined_graph)
stopwords = nltk.corpus.stopwords.words("spanish")
stopwords = [w.decode("utf-8") for w in stopwords]
```

Then we loop through all the nodes of the merged graph and add the stem nodes to each Spanish node. If the node has only one word (after stopword removal) we will use the NLTK stemmer; otherwise we just leave the phrase as it is:

In[59]:

```
combined_graph_stemmed = copy.deepcopy(combined_graph)
for node in combined_graph.nodes():
    if "lang" in combined_graph.node[node] and combined_graph.node[node]["lang"] == "spa":
        e = re.sub(" ?\([^)]\)", "", node)
        e = e.strip()
        stem = e
        words = e.split(" ")
        if len(words) > 1:
            words = [w for w in words if not w in stopwords or w == ""]
        if len(words) == 1:
            stem = stemmer.stem(words[0])

        stem = stem + "|stem"
        combined_graph_stemmed.add_node(stem, is_stem=True)
        combined_graph_stemmed.add_edge(stem, node)
```

Again we can count the nodes and the number of connected components. We see that the number of connected components decreases, as more nodes are connected into groups now:

In[60]:

```
networkx.algorithms.components.number_connected_components(combined_graph_stemmed)
```

Out[60]:

```
13944
```

In[61]:

```
len(combined_graph_stemmed.nodes())
```

Out[61]:

```
100447
```

Export the merged graph as DOT

The graph may now be exported to the DOT format, to be used in other tools for graph analysis or visualization. For this we use a helper function from the qlc library:

In[15]:

```
from qlc.translationgraph import read, write
OUT = codecs.open("translation_graph_stemmed.dot", "w", "utf-8")
OUT.write(write(combined_graph_stemmed))
OUT.close()
```

Extract a subgraph for the stem of "comer"

As an example how to further process the graph we will extract the subgraph for the stem "comer" now. For this the graph is traversed again until the node "com|stem" is found. All the neighbours of this node are copied to a new graph. We will also remove the sources from the node strings to make the final visualization more readable:

In[66]:

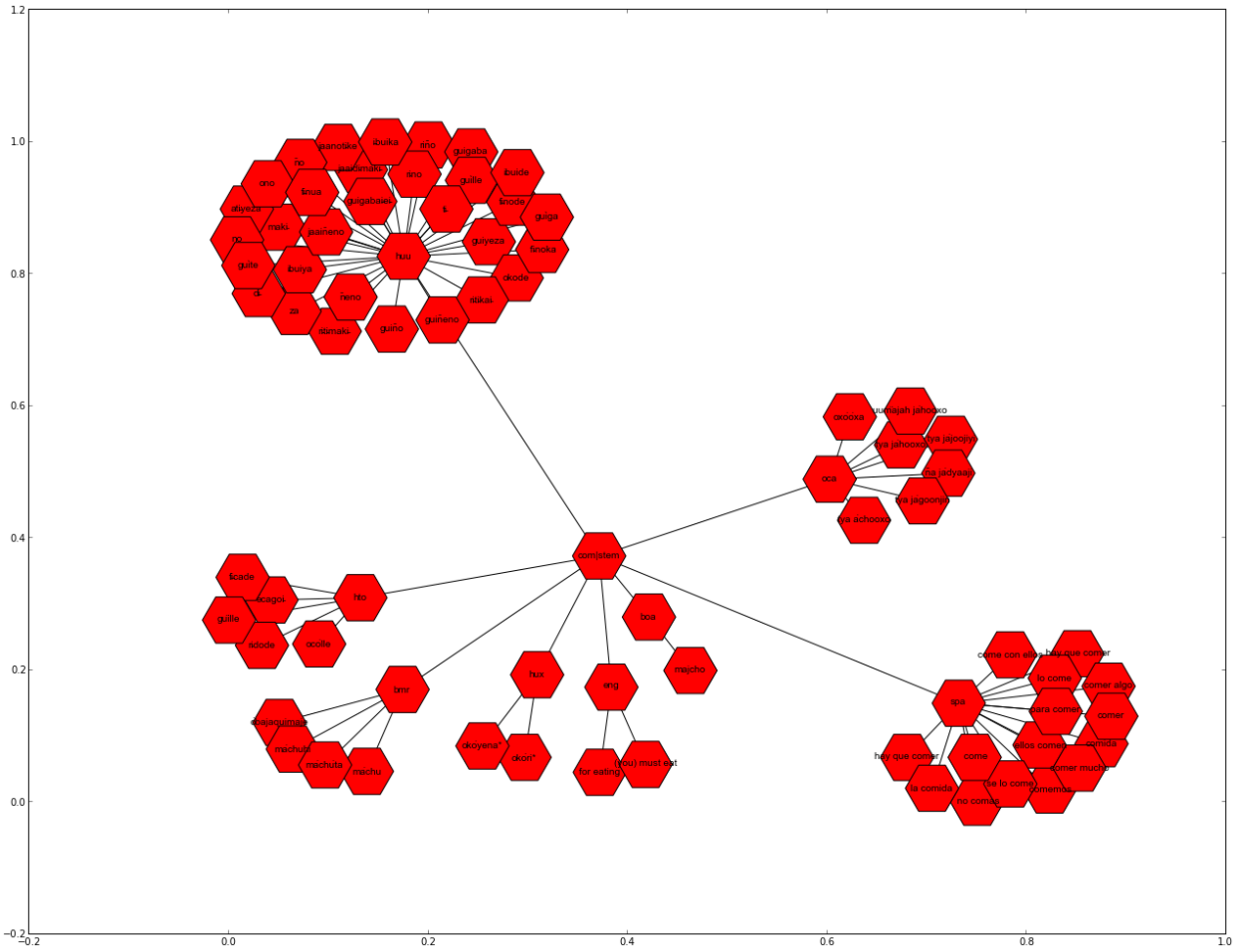
```
comer_graph = networkx.Graph()
for node in combined_graph_stemmed:
    if node == "com|stem":
        comer_graph.add_node(node)
        # spanish nodes
        comer_graph.add_node("spa")
        comer_graph.add_edge(node, "spa")

        for sp in combined_graph_stemmed[node]:
            if "lang" in combined_graph_stemmed.node[sp] and combined_graph_stemmed.node[sp]["lang"]
                comer_graph.add_node(sp)
                comer_graph.add_edge("spa", sp)

                for n in combined_graph_stemmed[sp]:
                    if ("lang" in combined_graph_stemmed.node[n] and combined_graph_stemmed.node[n]['
                            ("is_stem" not in combined_graph_stemmed.node[n] or not combined_graph_st
                        s = n.split("|")[0]
                        lang = combined_graph_stemmed.node[n]["lang"]
                        comer_graph.add_node(lang)
                        comer_graph.add_edge(node, lang)
                        comer_graph.add_node(s)
                        comer_graph.add_edge(lang, s)
```

Plot the subgraph with matplotlib

The subgraph that was extracted can now be plotted with matplotlib:

In[67]:

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize(22,17))
networkx.draw_networkx(comer_graph, font_family="Arial", font_size=10, node_size=3000, node_shape="H"
```



Export the subgraph as JSON data

Another method to visualize the graph is the D3 Javascript library. For this we need to export the graph as JSON data that will be loaded by a HTML document. The networkx contains a `networkx.readwrite.json_graph` module that allows us to easily transform the graph into a JSON document:

In[68]:

```
from networkx.readwrite import json_graph
comer_json = json_graph.node_link_data(comer_graph)
```

The JSON data structure can now be written to a file with the help of the Python `json` module:

In[69]:

```
import json
json.dump(comer_json, codecs.open("swadesh_data.json", "w", "utf-8"))
```

An example HTML file to visualize with D3 is here:

http://bl.ocks.org/4250342

1.2 GrAF/XML Data Sources

- MASC: http://www.anc.org/MASC/Home.html
- QuantHistLing: http://www.quanthistling.info/data

CHAPTER 2

graf-python Classes

2.1 Annotation

class graf.**Annotation**(*label*, *features=None*, *id=None*)

An Annotation is the artifact being annotated. An annotation is a labelled feature structure. The annotation class/interface also provides convenience methods for setting and getting values from a feature structure.

__init__(*label*, *features=None*, *id=None*)

Construct a new C{Annotation}.

Parameters

- **label** – C{str}
- **features** – C{list} of C{Feature} objects

2.2 AnnotationSpace

class graf.**AnnotationSpace**(*as_id*)

A collection of Annotations. Each AnnotationSpace has a name (C{Str}) and a type (C{URI}) and a set of annotations.

__init__(*as_id*)

Constructor for C{AnnotationSpace}

Parameters

- **name** – C{str}
- **type** – C{str}

remove(*ann*)

Remove the given C{Annotation} object.

Parameters **a** – Annotation

remove_where(*label*, *fs=None*)

Remove the C{Annotation}s with the given label in the given C{FeatureStructure}

Parameters

- **label** – C{str}
- **fs** – C{FeatureStructure}

2.3 Edge

class graf.**Edge**(*id*, *from_node*, *to_node*, *pos=None*)

Class of edges in Graph: - Each edge maintains the source (from) graf.Node and the destination (to) graf.Node. - Edges may also contain one or more graf.Annotation objects.

__init__(*id*, *from_node*, *to_node*, *pos=None*)

Edge Constructor.

id [str] The ID for the new edge.

from_node [graf.Node] The source node for the edge.

to_node [graf.Node] The target node for the edge.

pos [int, optional] An optional position of the edge in the graph. This will only be used when we render the graf, to make it easier to store an order of the edges.

2.4 FeatureStructure

class graf.**FeatureStructure**(*type_var=None*, *items=None*)

A dict of key -> feature, where feature is either a string or another FeatureStructure. A FeatureStructure may also have a type. When key is a tuple of names, or a string of names joined by '/', it is interpreted as the path to a nested feature structure. Additionally, a FeatureStructure defines the operations 'subsumes' and 'unify'.

__init__(*type_var=None*, *items=None*)

Constructor for C{FeatureStructure}.

Parameters **type** – C{str}

get_fs(*key*)

Returns the value corresponding to key if it is a FeatureStructure, and otherwise throws a ValueError

get_value(*key*)

Returns the value corresponding to key but throws a ValueError if it is a FeatureStructure

2.5 GrafRenderer

class graf.**GrafRenderer**(*outputfile*)

Renders a GrAF XML representation that can be read back by an instance of L{GraphParser}.

Version: 1.0.

__init__(*outputfile*)

Create an instance of a GrafRenderer.

render_ann(*a*)

Used to render the annotation elements of the Graph.

render_edge(*e*)

Used to render the edge elements of the Graph.

render_feature(*name*, *value*)

Used to render the features elements of the Graph.

render_fs(*fs*)

Used to render the feature structure elements of the Graph.

render_link(*link*)
Used to render the link elements of the Graph.

render_node(*n*)
Used to render the node elements of the Graph.

render_region(*region*)
Used to render the region elements of the Graph.

write_header(*g*)
Writes the header tag at the beginning of the XML file.

write_header_elements(*g*)
Helper method for write_header.

2.6 Graph

class graf.**Graph**
Class of Graph.

__init__()
Constructor for Graph.

create_edge(*from_node*, *to_node*, *id=None*)
Create graf.Edge from id, from_node, to_node and add it to this graf.Graph.

from_node [graf.Node] The start node for the edge.

to_node: graf.Node The end node for the edge.

id [str, optional] An ID for the edge. We will create one if none is given.

res [graf.Edge] The Edge object that was created.

find_edge(*from_node*, *to_node*)
Search for C{Edge} with its from_node, to_node, either nodes or ids.

Parameters

- **from_node** – C{Node} or C{str}
- **to_node** – C{Node} or C{str}

Returns C{Edge} or None

2.7 GraphParser

class graf.**GraphParser**(*get_dependency=None*, *parse_anchor=<type 'int'>*, *constants=<class 'graf.io.Constants'>*)
Used to parse the GrAF XML representation and construct the instance of C{Graph} objects.

version: 1.0.

__init__(*get_dependency=None*, *parse_anchor=<type 'int'>*, *constants=<class 'graf.io.Constants'>*)

parse(*stream*, *graph=None*)
Parses the XML file at the given path.

Returns a Graph representing the annotated text in GrAF format

Return type Graph

2.8 Link

class graf.**Link**(*vals=()*)

Link objects are used to associate nodes in the graph with the regions of the graph they annotate. Links are almost like edges except a link is a relation between a node and a region rather than a relation between two nodes. A node may be linked to more than one region.

__init__(*vals=()*)

2.9 Node

class graf.**Node**(*id=''*)

Class for nodes within a C{Graph} instance. Each node keeps a list of in-edges and out-edges. Each collection is backed by two data structures: 1. A list (for traversals) 2. A hash map Nodes may also contain one or more C{Annotation} objects.

__init__(*id=''*)

add_region(*region*)

Adds the given region to the first link for this node

clear()

Clears this node's visited status and those of all visited descendents

2.10 Region

class graf.**Region**(*id*, **anchors*)

The area in the text file being annotated. A region is defined by a sequence of anchor values.

__init__(*id*, **anchors*)

Constructor for C{Region}.

Parameters

- **id** – C{str}
- **anchors** – C{list} of C{Anchor}

2.11 StandoffHeader

class graf.**StandoffHeader**(*version='1.0.0'*, ***kwargs*)

Class that represents the primary data document header. The construction of the file is based on the ISO 24612.

__init__(*version='1.0.0'*, ***kwargs*)

Class's constructor.

version [str] Version of the document header file.

filedesc [ElementTree] Element with the description of the file.

profiledesc [ElementTree] Element with the description of the source file.

datadesc [ElementTree] Element with the description of the annotations.

2.12 FileDesc

class graf.**FileDesc**(***kwargs*)

Class that represents the descriptions of the file containing the primary data document.

__init__(***kwargs*)

Class's constructor.

titlestmt [str] Name of the file containing the primary data document.

extent [dict] Size of the resource. The keys are 'count' - Value expressing the size. And 'unit' - Unit in which the size of the resource is expressed. Both keys are mandatory.

title [str] Title of the primary data document.

author [dict] Author of the primary data document. The keys are 'age' and 'sex'.

source [dict] Source from which the primary data was obtained. The keys are 'type' - Role or type the source with regard to the document. And 'source'. Both keys are mandatory.

distributor [str] Distributor of the primary data (if different from source).

publisher [str] Publisher of the source.

pubAddress [str] Address of publisher.

eAddress [dict] Email address, URL, etc. of publisher. The keys are 'email' and 'type' - Type of electronic address, such as email or URL. Both keys are mandatory.

pubDate [str] Date of original publication. Should use the ISO 8601 format YYYY-MM-DD.

idno [dict] Identification number for the document. The keys are 'number' and 'type' - Type of the identification number (e.g. ISBN). Both keys are mandatory.

pubName [str] Name of the publication in which the primary data was originally published (e.g. journal in which it appeared).

documentation [str] PID where documentation concerning the data may be found.

2.13 ProfileDesc

class graf.**ProfileDesc**(***kwargs*)

Class that represents the descriptions of the file containing the primary data document.

__init__(***kwargs*)

Class's constructor.

catRef [str] One or more categories defined in the resource header.

subject [str] Topic of the primary data.

domain [str] Primary domain of the data.

subdomain [str] Subdomain of the data.

languages [array_like] Array that contains the codes of the language(s) of the primary data. The codes should be in the ISO 639.

participants [array_like] Array that contains the participants in an interaction. Each person is a dict element and the keys are 'age', 'sex', 'role' and 'id' - Identifier for reference from annotation documents. The 'id' key is mandatory.

settings [array_like] Array that contains the settings within which a language interaction takes place. Each settings is a dictionary and the keys are 'who', 'time', 'activity' and 'locale'.

add_language(*language_code*)
This method is responsible to add the annotations to the list of languages.

The language list in this class will represents the language(s) that the primary data use.

language_code [str] ISO 639 code(s) for the language(s) of the primary data.

add_participant(*id*, *age=None*, *sex=None*, *role=None*)
This method is responsible to add the annotations to the list of participants.

The parcipant list in this class will represents participants in an interaction with the data manipulated in the files pointed by the header.

A participant is a person in this case and it's important and required to give the id.

id [str] Identifier for reference from annotation documents.

age [int] Age of the speaker.

role [str] Role of the speaker in the discourse.

sex [str] One of male, female, unknown.

add_setting(*who*, *time*, *activity*, *locale*)
This method is responsible to add the annotations to the list of settings.

The setting list in this class will represents the setting or settings within which a language interaction takes place, either as a prose description or a series of setting elements.

A setting is a particular setting in which a language interaction takes place.

who [str] Reference to person IDs involved in this interaction.

time [str] Time of the interaction.

activity [str] What a participant in a language interaction is doing other than speaking.

locale [str] Place of the interaction, e.g. a room, a restaurant, a park bench.

2.14 DataDesc

class graf.**DataDesc**(*primaryData*)
Class that represents the annotations to the document associated with the primary data document this header describes.

__init__(*primaryData*)
Class's constructor.

primaryData [dict] Provides the location of the primary data document. The keys are 'loc' - relative path or PID of the primary data document, 'loctype' - Indicates whether the primary data path is a fully specified path (PID) or a path relative to the location of this header file, the default is 'relative', the other option is 'URL'. The other key is 'f.id' - File type via reference to definition in the resource header. All keys are mandatory.

add_annotation(*loc*, *fid*, *loctype='relative'*)

This method is responsible to add the annotations to the list of annotations.

The annotations list in this class will represents the documents associated with the primary data document that this header will describe.

loc [str] Relative path or PID of the annotation document.

fid [str] File type via reference to definition in the resource header.

loctype [str] Indicates whether the path is a fully specified path or a path relative to the header file.

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*