GraphAnno

An annotation and query tool
for graph-based linguistic annotations

Lennart Bierkandt

Friedrich-Schiller-Universität Jena
post@lennartbierkandt.de

Version of May 5, 2015

Contents

1 The annotation graph

1.1 Graph format

A graph in GraphAnno consists of a set of nodes and directed edges. Nodes and edges bear attributes in the form of key-value pairs, that are used for the linguistic annotation as well as the structuring and visualization of the graph. There are some attributes – like the attribute `type`, that differentiates the types of nodes and edges (annotation nodes, tokens, sentence nodes; annotation edges, ordering edges, section edges) – that serve the internal representation of the graph and are not edited directly. Under the key `attr`, however, there is a group of key-value pairs which serve the linguistic annotations and the representation of layers and which are edited by the user directly. They will be called *annotation attributes* in the following.

The graph in GraphAnno is segmented in sentences – units that that are used for structuring and displaying the corpus, but that do not necessarily correspond to real sentences (how ever these may be defined). There is a sentence node (bearing the attribute `type:s`) for every sentence, that bears information concerning the whole sentence. This includes, e.g., source, medium, speaker or the like. The annotations of this node is displayed in the light grey area below the sentence graph and under the text of the sentence (blue font) in black font. The information to which sentence a node of the graph belongs is represented by an (invisible) edge with `type:s` that links the node in question to the sentence node. Edges are not linked to a sentence node; their affiliation follows from the affiliation of their start and end nodes.

Token nodes are characterized by the attribute `type:t` and they carry the annotation attribute `token` with the token text as value. The order of the tokens of a sentence (as well as the sentence nodes themselves) is defined by ordering edges (with `type:o`) that link each node to its successor. These edges are needed for the correct visualization and for traversal; the are, however, not displayed and cannot be manipulated directly.

Among the annotation attributes, nodes and edges may bear the `cat` attribute, that has no special meaning for the graph, but which is desplayed prominently on top of the other annotations and without the key. Nodes and edges show numbers like `t23` for tokens, `n23` for other nodes and `e23` for edges, that are used for referencing the elements in annotation commands. These numbers are not stored in the graph model but generated dynamically each time the displayed graph is rendered.

GraphAnno also provides the possibility to assign nodes and edges to different layers. The fact that a node or edge is affiliated to a certain layer is represented by that element bearing an annotation (among the annotation attributes) with the key corresponding to the layer and the value `t` (for *true*). This representation allows for elements to belong to more than one layer, with the consequence that layers my overlap in an arbitrary manner. Nodes and edges of different layers may be displayed in different colors and aligned in different ways (hierachically or horizontally). Tokens do not belong to any layer; they are displayed in black by default.

1.2 Configuration

1.2.1 Layers and visualization

The window for configuration of the layers of the currently loaded graph and its visualization is opened with the commmand line command `config`.

In the section *general settings*, you can configure the settings for nodes and edges that do not belong to any layer.[1] *Default color* applies to all nodes and edges that are not tokens and that do not belong to any layer, *token color* applies to tokens, *found color* is used for the highlighting of nodes and edges found in a search, and *filtered color* for elements that are filtered out by the filter function. The setting for *edge weight* affects the layout of the displayed graph and shows its effect only when edges with different weight are present (details will follow in the next paragraph).

In the section *layers*, you can configure the layers of the graph. The *name* is an arbitrary label for the layer, that will be shown in the dropdown field for the layer selection. *Attribute* is the attribute that will be set to `t` for elements that belong to the layer in question. The *shortcut* is an identifier that can be used for the annotation of elements (cf. ??); this shortcut may only consist of alphanumeric characters and must not have the same structure as element references (i.e., `t`, `n` or `e` followed by a number, or `m`; cf. ??). *Color* means the color that is used for displaying the elements of the layer; *edge weight* is the weight of the layer's edges. The heigher this value (integer values), the shorter the rendering algorithm will try to make the edges. When you create two layers, one with a high edge weight and one with a low edge weight, the graph will be rendered such that the graph of the first layer is as compact as possible; the elements of the second layer will be placed in a way that they distort the first layer only to a low degree. If you enter `0` as edge weight, the edges will not enforce a hierachy between the nodes of that layer (otherwise, the start node of an edge is always placed higher than the end node). If a negative value is entered, all nodes of that layer are displayed horizontally on one tier.

The section *layer combinations* contains the settings for elements that belong to multiple layers. Via the checkboxes under *attributes* you determine, to which layers an element has to belong in order to be subject to the definition of the combination in question. The other settings work like those described for the *layers*, and for those elements that belong to the layer combination, they override the values given for the single layers.

Under *search makros* you can enter predefined graph-specific search makros as described in Section ??. These makros are then available for search queries in the graph. In the search makros text area, the definitions (`def ...`) have to be enterd each on its own line.

[1] Depending on the browser you use, the color fields are displayed as color picker or as simple text field. In the latter case, you have to provide the color as hexadecimal RGB value: A hash (`#`) followed by three two-digit hexadecimal numbers for red, green and blue, respectively. `#000000`, e.g., stands for black, `#ffffff` for white, `#ff0000` for light red etc.

1.2.2 Permitted annotations / tagset

In the window open by the command `tagste`, you can specify which annotation are permitted for the nodes and edges of the graph, i.e., you can define the graph-specific tagset. Only the keys and values defined here are permitted for the annotation of elements, illicit input is answered with an error message. Already existing annotations are not affected by changes of the permitted keys and values.

If you do not specify any keys and values, all annotations are permitted. If you specify keys, only these keys are allowed for annotations. If you leave the value field for a key empty, then all values are permitted for this key. If you want to restrict the possible values for a key, you may enter these into the value field of the key in question, separated by spaces.

The notation of the values is subject to the same rules as in the query language (cf. Section ??, p. ??): Simple values are entered without further markup; if a value contains special characters (siehe p. ??) it has to be enclosed in double quotes (`"..."`). Additionally – just like in the query language (p. ??) – you may use regular expressions, which are enclosed in slashes (`/.../`). In this case the regular expressions are anchored, i.e. an annotation value has to match the whole regular expression in order to be permitted.

1.2.3 Metadata

Additionally to the configuration of layers, visualization, search makros and permitted annotations, you may save metadata for a graph as key-value pairs. The command `metadata` opens the window, where you can enter an arbitrary number of keys with a text as corresponding value.

2 Keyboard shortcuts

In GraphAnno most of the functions related to navigation and display are controlled by keybord shortcuts. The following is a table of the available shortcuts:

Shortcut	Function
Navigation	
Alt + ⟵/⟶	previous/next sentence
Alt + Home/End	first/last sentence
Graph	
Ctrl + Shift + −/+	scale down/up graph
Ctrl + Shift + 0	zoom to fit (wrt. height)
Ctrl + Shift + arrows	move graph
Ctrl + Shift + Home/End	go to left/right edge of graph
Ctrl + Shift + Page up/Page down	go to upper/lower edge of graph
F4	toggle element references
Window	
F1	show/hide help window
F2	show/hide text and sentence annotations
F6	show/hide filter window
F7	show/hide search window

3 Command line commands

3.1 Data and navigation

3.1.1 Load file: `load`

With the command `load`, followed by the file name, you load a graph file inte the work space. Provide the file name without the extension `.json`; it has to enclosed in double quotes (`"..."`) if it contains spaces. Files are loaded from the `data` directory located in the GraphAnno main directory. Before loading, the work space is cleared from all data. So, changes that were not saved explicitly (using the command `save`) are lost. The name of the loaded file is shown next to the input line on the bottom of the user interface.

3.1.2 Load file: `add`

Just like the command `load`, the command `add` loads a file into the work space. The difference is, that the work space is not cleared – the new file is added and the new sentences are appended to the existing ones. After adding the file no file is shown as loaded next to the input line and the files cannot be saved separately anymore.

3.1.3 Save file: `save`

The command `save` saves the work space to a GraphAnno file in the `data` directory. The file name has to be entered in the same way as with the command `load`. If there is a filename indicated next to the input line, the work space can be saved to this file without specifying the file name. Attention: no warning is issued when an existing file will be overwritten.

3.1.4 Clear work space: `clear`

The command `clear` clears the work space from all date. Changes that were not saved are lost. Next to the input line no file will be indicated anymore.

3.1.5 Create new sentence: `ns`

With the command `ns` – followed by one or more sentence names separated by spaces – you can create a new sentence. The command creates sentence nodes with the corresponding `name` attribute; afterwards you are directed to the first newly created sentence.

3.1.6 Delete sentence: `del`

The command `del` deletes the current sentence including the sentence node.

3.1.7 Gehe zu: `s`

In order to navigate from sentence to sentence, you may use (alternatively to keyboard shortcuts or the dropdown field) the command `s`, followed by the name of the sentence you want to change to. You are then directed to the first sentence with the given name.

3.1.8 Export graphics: `image`

With the command `image`, you can export the graphic that GraphAnno is displaying for the current sentence. The first argument is the desired format. All formats supported by Graphviz are available, e.g. `dot`, `eps`, `pdf`, `png` or `svg`. See under `http://www.graphviz.org/content/output-formats` for the complete list. The second argument is the name of the new image file (without extension; put the name in double quotes if it contains spaces). The image will be saved in GraphAnno's `images` directory.

3.1.9 Export corpus: `export`

Using the command `export`, you can export the contents of the work space as a corpus in another format. The first argument is the format (at present the only fully functional format is `sql` for the import in GraphInspect; the format `paula` is theoretically available, but it is heavily restricted with respect to layers), the second argument is the name of the corpus to be saved. For `paula` you can optionally specify the name of the corpus document to be created as third argument. The exported corpus will be saved in GraphAnno's `exports` directory.

3.1.10 Import text: `import text`

You can import texts with the command `import text`. After issuing the command, a window opens where you can enter the text and set the preferences for its processing. The work space will be cleared before the text is imported (but not as soon as the window opens). Changes that hadn't been saved are lost and no file is indicated next to the input line anymore.

In the import window you can choose between two methods of entering your text: You can upload a text file or you can paste it in the text area. For the processing of the text there are two methods available, too. For unedited text you may use the method "punkt segmenter". This method uses an automatic segmenter to split the text into sentences and tokens. In order to process abbreviation etc. correctly, you need to specify the language of the text.

The second processing method is "regular expressions"; this method is made for preformatted texts. First, you have to enter a string that will be used for segmenting the sentences. The preset is `\n` [2] for a file in which every sentence starts on a new line. The second string you have to enter a a regular expression that matches the tokens. The preset here is `(\S+)`. This stands for a sequence of non-spaces, so all words that are separated by spaces are matched as tokens. The purpose of the parentheses is to save the matched string in the variable `$1`, so it can be used in the next field. The next field is for an annotation command (see ??) for the tokens, that uses the string matched by the regular expression in the preceding field. The preset here is `token:$1`. That means that the string matched as token is used for the annotation of the token text. Another example would be a text tagged for parts of speech, where the part of speech is appended to every word with an underscore. In this case you would enter the regular expression `(\S+)_(\S+)` and the annotation command `token:$1 pos:$2`. The regular expression in this case finds two strings of non-spaces that are joined by an underscore; the strings are saved to the variables `$1` (the word) and `$2` (the POS tag). In the annotation command these variables are used to annotate the token text and the `pos` attribute.

tbc.

[2] `\n` stands for a line break, `\t` for a tab.