GraphAnno

An annotation and query tool
for graph-based linguistic annotations

Lennart Bierkandt
Friedrich-Schiller-Universität Jena
post@lennartbierkandt.de

Version of August 18, 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. 3.2.1 or ??); this shortcut may only consist of alphanumeric characters and underscores and must not have the same structure as element references (i.e., `t`, `n` or `e` followed by a number, or `m`; cf. 3.2.3). *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

[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.

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.

1.2.2 Permitted annotations / tagset

In the window open by the command `tagset`, 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 (see 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 Annotation makros

For a graph you save annotation makros, that facilitate the annotation with frquently needed attribute combinations. With the command `makros` you open a window where you can define these makros. In the fields on the left you enter the shortcut for your makro (it has to consist of alphanumeric characters including the underscore, and it

should not have the form of an element reference or be identical to a layer shortcut, cf. Section 1.2.1); in the corresponding field on the right you enter the desired annotations for your makro. For these annotations you must use the same syntax as in the annotation commands, i.e. a set of attributes in the form `key:value`, separated by spaces (see Section 3.2.1 for details).

1.2.4 Metadata

Additionally to the configuration of layers, visualization, 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 data. 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. If you enter one or more sentence names as arguments, it is not the current sentence that will

be deleted but the sentences that bear one of the given names. A further possibility is to enter a regular expression (in slashes); in that case, all sentences whose names match the given regular expression will be deleted.

3.1.7 Set sentence: `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 3.2.3) 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.

3.1.11 Import Toolbox data: `import toolbox`

Toolbox files can be imported using the command `import toolbox`. This command opens a window in which you can choose the file to be imported and enter the format description. The format description has to be in JSON format and consists of a list of a list of markers. The lists are sorted according to their levels – the highest (*record* level) first – and contain the markers that belong to the respective level (markers are entered without backslash). The first marker of the first level (`ref` in the example below) will be used as record ID. The marker whose line is to be used as token text is preceded by an asterisk. Elements that lie below the token level will be joined and integrated into their respective tokens.

A format description for a toolbox file with three levels (record, word, morpheme) could look like this, e.g.:

```
[["ref", "eng"], ["*gw"], ["mph", "ge", "ps"]]
```

Like with the command `load`, the work space will be cleared when importing. Changes that haven’t been saved will be lost; next to the input line no file will be indicated anymore.

3.1.12 Export and import configuration: `export` and `import`

The command `export` serves also for exporting graph configurations, that can be imported in other graphs using the command `import`. As first argument you enter the

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

type of the configuration to be exported or imported: `config` for layers and visualization configuration (see 1.2.1), `tagset` for permitted annotations / tagset (see 1.2.2). The second argument is the filename for the configuration file to be saved or to be imported (without file extension). The exported file will be saved in a subdirectory – named like the configuration type – of the `exports` directory. Attention: when importing, the existing configuration will be replaced completely.

3.1.13 Edit configurations: `config`, `tagset`, `makros` and `metadata`

These commands open the windows for the settings that were described in Section 1.2. `config` for layers and visualization, `tagset` for permitted annotations / tagset, `makros` for annotation makros and `metadata` for metadata.

3.2 Annotation commands

GraphAnno's annotation commands are designed to be entered quickly, so their syntax is rather compact: they consist of a short command (often one letter only) and are followed by parameters seperated by spaces.

The commands require to be in a sentence, i.e., you have to create a sentence first (using the command `ns`, see 3.1.5) if the work space is empty.

3.2.1 New node: `n`

The command for creating a new node is `n`, followed by the attributes the new node is to bear as key-value pairs in the format `key:value`. Key and value can be given either as simple string (if it doesn't contain any of the special characters used in the annotation languages: `␣:"#`[3]) or as string in double quotes (`"..."`), that may contain any character (double quotes themselves have to be excaped with a backslash: `\"`).

You can also use the shortcut from your previously defined annotation makros (see Section 1.2.3). When you additionally enter attributes with keys present in the makro, these override the annotations defined in the makro.

Additionally, you can specify the layer to which the new node is to belong (if you don't, it will belong to the layer set in the layer dropdown field). For this purpose you use the shortcuts defined in the layer configuration (cf. Section 1.2.1). The use of these shortcuts also has the effect of a switch, insofar as it sets the layer for the following operations (like the command `l`, see 3.2.10).

Command `n` in modified BNF:

[3] The symbol ␣ stands for the space.

```
command_n           =   "n " attributes
attributes          =   attributes " " attributes
                        attribute
                        annotation_shortcut
                        layer_shortcut
attribute           =   key string
key                 =   string ":"
string              =   character_not_special+
                        """ character* """
alnum               =   letter | digit | "_"
annotation_shortcut =   alnum+
layer_shortcut      =   alnum+
```

3.2.2 New edge: `e`

The command for creating a new edge is `e`, followed by start and end node of the edge to be created and the attributes, it is to bear. Like with `n`, the specification of a layer is possible.

Command `e` in modified BNF:

```
command_e       =   "e " start_end " " start_end " " attributes
start_end       =   node_reference
                    token_reference
node_reference  =   "n" number
token_reference =   "t" number
```

3.2.3 Annotieren: `a`

The command for annotation elements is `a`, followed by the elements to be annotated and the attributes with which they are to be annotated (all given elements will be annotated with all given attributes; also layer shortcuts can be used). The order of the elements and attributes is free. You can also annotate sequences of elements of the same type (i.e., `n`, `e` or `t`) by entering the first and the last element joined by two dots. E.g., when you enter `t3..t7`, all tokens from `t3` to `t7` will be annotated (you may enter the sequence also in the inverse order, i.e., `t7..t3`).

At the same time you can use the command `a` for deleting attributes. In order to do so, enter the key to be deleted with colon, but without value.

Command `a` in modified BNF:

```
command_a           =   "a " a_parameters
a_parameters        =   a_parameters " " a_parameters
                        element_reference
                        attributes
                        key
element_reference   =   node_reference
                        edge_reference
                        token_reference
                        meta_node_reference
                        element_sequence
edge_reference      =   "e" number
meta_node_reference =   "m"
element_sequence    =   node_sequence
                        token_sequence
                        edge_sequence
node_sequence       =   node_reference ".." node_reference
token_sequence      =   token_reference ".." token_reference
edge_sequence       =   edge_reference ".." edge_reference
```

3.2.4 Delete elements: `d`

Elements are deleted with the command `d`, followed by the elements to be deleted. If you delete nodes, the outgoing and ingoing edges are deleted as well. If you delete a token node from the middle of a sentence, the adjacent tokens are joined automatically.

Command `d` in modified BNF:

```
command_d       =   "d " d_parameters
d_parameters    =   d_parameters " " d_parameters
                    element_reference
```

3.2.5 Group nodes under new parent node: `g` oder `p`

The grouping command `g` or `p` creates a new parent node for the given nodes. I.e., the command creates a new node and edges that connect the new nodes to the nodes to be grouped. The parameters of this command are the nodes to be grouped and the attributes the newly created node is to bear. The order of nodes and attributes is irrelevant. Like with the command `n`, a layer can be specified.

Command `g/p` in modified BNF:

```
command_g           =   ("g " | "p ") g_parameters
g_parameters        =   g_parameters " " g_parameters
                        node_reference
                        token_reference
                        node_sequence
                        token_sequence
                        attribute
                        layer_shortcut
```

3.2.6 Append child node: `h` or `c`

The command `h/c` works analogously to the command `g/p`, but instead of a parent node a new common child node is created.

Command `h/c` in modified BNF:

```
command_h           =   ("h " | "c ") h_parameters
h_parameters        =   h_parameters " " h_parameters
                        node_reference
                        token_reference
                        node_sequence
                        token_sequence
                        attribute
                        layer_shortcut
```

3.2.7 Insert node into edge: `ni`

The command `ni` (`node insert`) allows you to insert a new node into an existing edge. As parameters you specify the edge and the attributes for the new node. A new node will be created, and the given edge will be replaced by two edges with the same annotations, that connect the start node of the original edge with the new node and the new node with the end node of the original edge.

If you specify more than one edge, a new node will be inserted into each of them

Command `ni` in modified BNF:

```
command_ni          =   "ni " ni_parameters
ni_parameters       =   ni_parameters " " ni_parameters
                        edge_reference
                        attributes
                        layer_shortcut
```

3.2.8 Delete node but preserve connections: `di` und `do`

If you want to delete a node but preserve the connections between parent node(s) and child node(s) of the deleted node, you can use the commands `di` or `do`. These commands delete the specified node and connect each child node to each mother node (this makes sense particularly in a tree-like structure where there is one parent node and many child nodes). `di` (*delete ingoing*) deletes the ingoing edge(s), `do` (*delete outgoing*) deletes the outgoing edge(s).

Commands `di` and `do` in modified BNF:

```
command_di_do       =   "di " node_reference+
                        "do " node_reference+
```

3.2.9 Tokenize: `t`, `tb`, `ta`

For creating tokens there are the commands `t`, `tb` and `ta`. The arguments for these commands are a list of words, separated by spaces. These words are inserted as tokens into the current sentence. If the sentence already contains tokens, the command `t` appends the new ones. `tb` (`tokenize before`) and `ta` (`tokenize after`) take a token as their first argument and insert the new tokens before or after, respectively.

The words can be given as bare strings, or, if they contain control characters (␣:"#), as string in double quotes (`"..."`; double quotes inside the string have to be escape with a backslash: `\"`).

Commands `t`, `tb`, `ta` in modified BNF:

```
befehl_t        =   "t " words
words           =   words " " words
                    word
word            =   non-control-character+
                    """ character* """
```

```
command_tb      =   "tb " token_reference " " words
command_ta      =   "ta " token_reference " " words
```

3.2.10 Set layer: `l`

As an alternative to the select field, you can set the layer used for the subsequently created elements with the command `l`. For the layers you use the shortcuts defined in the layer configuration (cf. 1.2.1).

Command `l` in modified BNF:

```
command_l   =   "l " layer_shortcut
```

tbc.