The Dependency Synthesisier Manual

Michael Daum

Natural Language Systems
Department of Informatics
University of Hamburg

micha@nats.informatik.uni-hamburg.de

October 20, 2004

Contents

1 SYNOPSIS

Depsy

[**-annos** ***filename***]
[**-annoprefix** ***string***]
[**-debug**]
[**-debug-label-table**]
[**-extralevels** ***string***]
[**-index** ***number***]
[**-input-format|if** ***string***]
[**-levels** ***filename***]
[**-lexicon** ***filename***]
[**-man**]
[**-max|n** ***int***]
[**-output-format|of** ***string***]
[**-plugin** ***filename***]
[**-prefix** ***string***]
[**-range "*range-spec*"**]
[**-syntax|help|?**]
[**-wordgraphs** ***filename***]
[***inputFile1 inputFile2 ...***]

2 DESCRIPTION

This tool is part of the CDG parsing suite. It translates annotations given in the negra file format to cdg annotations, and thereby translate phrase structure representations into dependency representations of the annotations. Additionally it can generate a rudimentary lexicon and a level description in order to load the generated data into the WCDG system.

For debugging purpose the phrase structures extracted from the negra format are converted to a format resembling the Penn Treebank format which is much easier to read. Note, that this output is not thought to be correctly with regards to the Penn Treebank format definition but only to resemble to it. At least there is additional information written, i.e. the node number linking back to the lines in the negra format sentence (0-499 for leaves, 500- for inner nodes). The head phrases analysed are marked as well using `*` at the lexical level and `!!` for the inner nodes (see the function `markLexHeads` §5.3.41).

There are a couple of configurations which are possible in order to customize the desired translation. Actually reducing the negra phrase structures to dependency structures is not only a process of information reduction but also enriching the annotation with information needed with respect to the target corpus. And last not least, not all phrase structures are annotated with lexical heads.

3 OPTIONS

`Depsy` reads from STDIN or from the specified files (*inputFile1 inputFile2* ...) and stores the results in one or more files in the format specified by **--output-format** (default 'cdg'). To configure that tool you have to write a plugin that you specify with the **-plugin** option. Results of are written to the files that you specify with **-annos**, **-lexicon**, **-levels** and **-wordgraphs**. There are two flavours of output format: 'plain' and 'cdg'. The latter format is valid CDG input to be used with the CDG parser developed by the same team. In general, output-files that aren't specified have no defined default value and thus no corresponding information is extracted. In what follows you get a detailed list of all those parameters.

-annos *filename*

generate dependency trees and save them in *filename*.

-annoprefix *string*

(default 'auto-s')

This option lets you specify the prefix string to be added to the annotation's name.

-debug

switch on the debugging mode. The currently processed sentence and other stuff is displayed on `STDERR`.

-debug-label-table

Display statistics about the label table. For each rule a sample of the sentences in which the rule was used is given, plus the total number of applications.

-extralevels *string*

This option can be set to a comma-separated string. It will cause the output of Depsy to contain other levels than the SYN level. All subordinations on these levels will be unlabelled NIL bindings, unless you change them via dependency conversion rules.

-index *number*

specifies the sentence index which we start to number wordgraphs and annotations. Note, that the Negra Treebank numbers sentences by itself which has higher priority than the index that we might want to have giving this index option. This is actually only usefull if there is no explicite sentence numbering scheme and we have to deal with that ourselfts as it is in the Penn Treebank.

-input-format *string*

TODO (default 'negra')

-levels *filename*

generate a syntax level declaration including all needed labels. A level declaration is going to be generated only if a *filename* is specified.

-lexicon *filename*

extract a rudimentary lexicon from the morphological annotations. This basically serves the purpose to let you load the generated annotations into your xcdg to display the annotations as graphical trees. If you don't specify an *filename* you will not get a lexicon.

-man

print out the complete manual of `Depsy`.

-max|n *int*

specify the maximum number of sentences that are converted. If no maximum is specified via this parameter we extract as many as possible.

-output-format *string*

TODO (default 'cdg')

-plugin *filename*

TODO

-prefix *string*

TODO (default 's')

specify the prefix string to be used when naming annotations and wordgraphs.

-range "*range-spec*"

specify a list of intervals of annotation ids to be processed. You might specify an interval 1-625, which includes all annotations between 1 and 625 or any arbitrary numbers, you might say -400 specifying the first 400 annotations, or 10000- to process the annotations 10000 til the end. You can combine comma separated a list of intervals, e.g. 1-10,50-70,1000-

-syntax|help|?

a short syntax summary message.

-wordgraphs *filename*

generate wordgraphs as linear cdg sentences in *filename*. If you don't use this option you will not get the cdg sentences.

4 CONFIGURATION

4.1 Writing plugins

A plugin basically must provide ... tabels that configure the behavior of `Depsy`. These are:

%headTable

The purpos of this hashtable is to establish rules to detect lexical heads of a phrase structure, an information which is not annotated in most cases. This information is applied by the function `markLexHeads` §5.3.41.

%phraseConvTable

This hashtable maps node tags to a list of perl functions the purpose of which is to always call these functions in the given order whenever we come past the node tag. This happens in the function `convertPhraseTree` §5.3.40 after the recursive phrase structure has been build and before the dependency structure is extracted from that.

@labelTable

This is a priority list of in order to configure the label generation of the dependencies. The `@labelTable` is used by the function `makeDependency` §5.3.47.

%depConvTable

This hashtable maps dependency labels to a list of perl functions the purpose of which is to always call these functions in the given order whenever we come past the named label. This happens in the function `convertDepTree` §5.3.39 after the dependency tree has been constructed.

%callbacks

This is the hash of known callbacks. Below is the list of all known callbacks. Each callback points to a function which returns 0 or 1 depending on success or failure. Every known callback is sensitive for this return value.

`readSentence(nodes)`
: called at the end of §5.3.12 after one sentence (between BOS and EOS) has been parsed. `nodes` is a pointer to the hash to all nodes that have been generated for that sentence. If this callback returns 0, processing of that sentence is aborted, and the next one is read in.

`preProcess(edges)`
: Called before processSentence operates. The parameter is a pointer to the root of the phrase tree.

`postProcess(edges)`
: called at the end of §5.3.4. `edges` is a pointer to the list of all dependency edges that have been constructed. The return code of the callback becomes the return code of §5.3.4, that is treebank conversion is aborted on 0 and continued on 1;

`initialize()`
: called before starting to read the treebank.

`finalize()`
: called at the end of the §5.3.54 routine.

`writeLexiconEntry`
: called before writing a lexical entry in §5.3.26. This callback gets a tree node as a parameter. See §5.3.25 on how to generate some default lexicial features for the lexicon entry.

  If this callback returns 0, then no lexical entry is printed. Maybe you wanted to suppress some of these, maybe you generate lexical entries by yourself in the `LEX` stream. Otherwise 1 should be returned.

...

5 IMPLEMENTATION

5.1 Global Variables

5.1.1 Variables set by command line parameters

All of the following variables are initialized by command-line parameters as far as they are given in a specific call, leaving some variables undefined.

`$debug` (default 0)

This variable holds the debugging mode with values 0 or 1 switched on/off by command-line parameter `-debug` as described in section §3. In debug mode all debug messages are generated to STDERR.

`$maxSentence`

This variable is an integer value greater zero set by command-line parameter `-max` (or `-n`). No more than `$maxSentence` are generated by `$Depsy`.

`@sentenceIdRange`

range-specification as described above for the command-line parameter `-range`

`$wordgraphsFile`

This variable holds the filename where the constructed wordgraphs should be written in. See `-wordgraphsFile` in section §3

`$lexiconFile`

This variable holds the filename of the lexicon to be constructed. See `-lexicon` in section §3

`$annoFile`

This variable holds the filename of the resulting annotations in cdg format. See `-annos` in section §3

`$levelDclFile`

This variable holds the filename of the level declaration which might optionally be build. See `-levels` in section §3 undef

`$pluginFile`

This is the filename of the plugin to be loaded.

`$inputFormat`

This Variable holds the format of the input to be read. Valid values are "penn" and "negra".

`$outputFormat`

This Variable holds the format of the output to be written. Valid values are "cdg" and "plain".

`%wgPrefix`

This is the prefix string added to the names of the generated wordgraphs.

`%annoPrefix`

This is the prefix string added to the names of the generated annotations. wordgraphs.

5.1.2 Plugin tables

`%headTable`

See section Writing plugins.

`@labelTable`

See section Writing plugins.

`%phraseConvTable`

See section Writing plugins.

`%depConvTable`

See section Writing plugins.

`%callbacks`

See section Writing plugins.

5.1.3 Variables used in the conversion processes

`$rootNode`

pointer to the current rootNode of the phrase tree.

`%nrSentences`

number of sentences processed so far. Its value is increased by processSentence only.

`%sentenceIndex`

current index of the sentence that is being processed. You can set the initial value using option `-index`.

`$extralevels`

undocumentend feature

`@extralevels`

undocumentend feature

5.1.4 Storage for the negra sections

The following variables contain the eight negra tables `ORIGIN`, `EDITOR`, `WORDTAG`, `MORPHTAG`, `NODETAG`, `EDGETAG` and `SECEDGETAG` as they are read from the input.

`@origins`

This list contains the `ORIGIN` negra table and contains the name and the comment field of an origin-id. See the function `readOrigins` §5.3.5.

`%editors`

This hashtable maps editor ids to their login and their name. Actually this information isn't used anywhere. But as long as the negra sources contain this info we eat them. See the function `readEditors` §5.3.6.

`%wordTags`

This hashtable maps a tag to its description. The rest of the available information, i.e. the id and the flag, aren't analyzed, besides the fact that the actually used word tags in the sentences aren't checked for existence in this table. See the function `readWordTags` §5.3.7.

`%morphTags`

This hashtable maps the morph tag ids to their realization. the `%morphTags` are used for generating the features of a word in an annotation in `getFeatures` §5.3.25. See the function `readMorphTags` §5.3.8.

`%nodeTags`

This hashtable maps the node tag to its human readable description, e.g. `"NP" => "noun phrase"`. It is of no actual use here also. See the function `readNodeTags` §5.3.9.

`%edgeTags`

This hashtable maps the edge tag to its human readable description, e.g. `"NK" => "noun kernel modifier"`. Same usefulness as the above `%nodeTags` (none). See the function `readEdgeTags` §5.3.10.

`%secEdgeTags`

TODO

`%lexicon`

TODO

`%labels`

TODO

`$nolabelWarning`

this flags the 'no label' warning having occured

5.2 Data Structures

In this section we describe the most important data structures as they are used in the implementation. Some data structures have already been described in the section §5.1 as they are global variables used to store the negra input. But after that we want to discuss at least the main data types that we are going to stumble across in section §5.3. This is

- a *linear representation*

  of the phrase structures as they are read in from the negra input file, called `%sentence` (§5.2.1),

- which is the converted into a *recursive data type*
  made out of so called `%nodes` (§5.2.2),
- and which are then converted into a list of *dependency edges*
  that is some `@deps`, a list of dependencies (§5.2.3).

5.2.1 A linear phrase structure representation

TODO
See `readNegraSentence` §5.3.12 and `makePhraseTree` §5.3.38.

5.2.2 A recursive phrase structure representation

TODO: elaborate!!!

```
tree = {
  parent => tree
  children => <list of trees>,
  nodeTag => <nodeTag>,
  edgeTag => <edgeTag to parent>
  id => <index of word in the sentence>
}
```

5.2.3 Dependency structures

5.3 Functions

5.3.1 `debug()`

print debug messages if the `$debug` flag is set.

5.3.2 `readNegraTreebank()`

This functions reads and processes the input which is suppposed to be in Negra Export format.

5.3.3 `readPennTreebank()`

This functions reads and processes the input which is suppposed to be in Penn Treebank format. It therefore uses a RecDescent grammar which calls processSentence for each sentence.

5.3.4 `processSentence()`

This function is called whenever one sentence has been read from the input and a recursive phrase data structure has been built in memory.

5.3.5 readOrigins()

5.3.6 readEditors()

5.3.7 readWordTags()

5.3.8 readMorphTags()

5.3.9 readNodeTags()

5.3.10 readEdgeTags()

5.3.11 readSecEdgeTags()

5.3.12 readNegraSentence()

5.3.13 escapeIdentifier()

escape and quote a string only if necessary

5.3.14 writeWordgraph()

5.3.15 mapCase()

5.3.16 mapGender()

5.3.17 mapNumber()

5.3.18 mapPerson()

5.3.19 mapTense()

5.3.20 mapDegree()

5.3.21 mapMood()

5.3.22 mapDefinite()

5.3.23 mapFlexion()

5.3.24 isIncluded()

check if a number is included in a list of intervals, e.g. 1-2,5-10,12,18-

5.3.25 getFeatures()

5.3.26 writeLexiconEntry()

5.3.27 writeLevelDecl()

This function creates the level declaration if you specified the **-levels** *filename*. By default one SYN level is constructed marked to be the mainlevel. It declares all labels that have been found during tree conversion. (see %labels).

5.3.28 isLeaf()

Return true if a node in a phrase tree is a leaf node.

5.3.29 `formatPhraseTree()`

Format a phrase tree like this: [S [NP das][VP ist [ADJP gut]]] (but with more newlines)

5.3.30 `printPhraseTree()`

Print bracketed representation of phrase tree on STDERR.

5.3.31 `computeExtents()`

Compute the linear extent of each constituent and propagate the info to $$x{min} and $${max} for each node.

Leaf nodes have min == max == id. Inner nodes have the smallest interval that encloses all leaves. This means that interleaved constituents have extents that partially overlap.

5.3.32 `insertInnerNode()`

Insert a new inner node for TREE under PARENT with the values specified in ATTS.

5.3.33 `spliceWords`

Manipulate phrase tree $tree so that leaves $x through $y turn into a new word with the specified parameters.

5.3.34 `splitWord()`

Manipulate phrase tree $tree so that the leaf $w turns into $n new words.

5.3.35 `moveNode()`

Manipulate phrase tree $tree so that $node is no longer a child of its former parent, but a child of $new_parent, all parent/child relations are consistent again, extents are updated, and everybody is happy.

5.3.36 `deleteNode()`

Manipulate phrase tree $tree so that $node no longer exists.

5.3.37 `moveEdge()`

Manipulate a dependency tree so that edge $e points to $p, and the ->children arrays are consistent.

5.3.38 `makePhraseTree()`

This function is used by readNegraTreebank() to construct a nested phrase data structure from a linear sentence hash.

5.3.39 `convertDepTree()`

manipulate a dependency tree

5.3.40 `convertPhraseTree()`

manipulate an annotated phrase tree

5.3.41 `markLexHeads()`

5.3.42 `getMinId()`

5.3.43 `getAllNodes()`

5.3.44 `writePhraseTree()`

5.3.45 `makeAllDeps()`

wrapper for makeDeps: Magerman/Lin differences: - they dont construct NIL dependencies - they can handle only a single rooted phrase tree / dependency tree - we can optionally construct more than one level of description

5.3.46 `makeDeps()`

see (Magermann 1994, p.64-66) and (Lin 1995) returns the lexical head of the analysed node as a side effect new dependency edges are asserted into deps

5.3.47 `makeDependency()`

5.3.48 `getDominator()`

given two nodes n1 and n2 of a phrase tree return the ancestor of n1 which dominates n2

5.3.49 `findHeadChild()`

5.3.50 `writeAnnotation()`

5.3.51 `formatNodes()`

5.3.52 `makeSnapShot()`

Create a PNG image of the current state of the phrase structure tree. This is used strategically during phrase tree conversion to leave a trail of how the transformation went.

5.3.53 `depSnapShot()`

5.3.54 `main()`

This is the first function called comparable to the `main()` function in C and elsewhere. It reads all command line parameters, opens the configuration files, creates the initially empty output files and loops over all input that is given to `Depsy`.