

Documentation of the
dopdis
Parsing Environment

Version number 0.1 (2004)

Khalil Sima'an Andreas Zollmann
simaan@science.uva.nl zollmann@gmx.de

Institute for Logic, Language and Computation
University of Amsterdam
January 28, 2005

Contents

1	Copyright Issues	4
2	Installation	4
2.1	The Contents of <code>dopdis.tgz</code>	4
2.2	Preparing the environment	5
3	Getting Started: An Example	6
3.1	Working with DOP1	7
3.1.1	Splitting the corpus	7
3.1.2	DOP1 Training	7
3.1.3	Running the parser	9
3.1.4	Evaluating the parser	10
3.2	Working with DOP*	10
3.3	Tips	10
3.4	What's next?	11
4	Trebank Format and Viewing	11
5	Training <code>dopdis</code>: Basic Programs	12
6	The DOP* estimation variant	13
6.1	Invocation	14
6.1.1	Training mode	14
6.1.2	Testing mode	15
6.2	Settings	15
6.2.1	Corpus type	15
6.2.2	Using <code>mysql</code> for DOP* training	15
7	Setting the pruning parameters for <code>dopdis</code>	16
7.1	Kind of pruning and limitations	16
7.2	Parameter details	16
8	Further Documentation	17
9	Credits	18

10 Acknowledgments	18
A Details on major dopdis programs	19
A.1 Parameters of gen-t-grams	19
A.2 The arguments of dopdis	20
A.3 The input format for dopdis	22

1 Copyright Issues

The **dopdis** environment is free software. You can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

The programs are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License (file `dop/doc/gpl.txt`) for more details.

2 Installation

The **dopdis** environment has been designed to run on a Linux platform. For other Unix systems, perhaps some code has to be modified in order to get certain programs running. Also, make sure that you've got decent versions of GNU C, Java, Perl, and Python installed.

2.1 The Contents of `dopdis.tgz`

When decompressing the installation archive `dopdis.tgz`, the following directories will be created:

dop/bin Contains executable binaries.

You should add this directory to your `$PATH` environment variable in order to be able to execute the available training programs¹

dop/source Contains the C source files.

This directory is not necessary for training and creating a `dopdis` parsing under Linux. These are only the sources for the parsing algorithms.

dop/source_evalb Contains the source files and documentation for *evalb*, written by Satoshi Sekine and Mike Collins.

dop/source_jdbm Contains the source files for the java package JDBM, maintained at <http://jdbm.sourceforge.net/>.

dop/objects Object files which are used during each training process when training and compiling a new `dopdis` parser: the object files are linked with the other

¹In general, you can add any directory, say `/home/simaan/bin` to your path by adding the following line at the end of your `.cshrc` file in your home directory: `setenv PATH ${PATH}:/home/simaan/bin` After adding this line, save `.cshrc` and execute `source .cshrc`.

code that represents the probabilistic grammar extracted from the training tree-bank.

dop/temp Directory for temporary files, used by the java programs. The programs keep the temporary files for inspection after execution. If disk space on the current file system is scarce, `dop/temp` should be used as an anchor to a directory of another file system.²

dop/scripts Contains executable scripts.
Add this also to your `$PATH` env variable.

dop/javastuff Contains the `.java` and `.class` files for corpus splitting, parser evaluation, and DOP* training/testing. Contains also the *python* scripts called by java class `DOPStar` as well as several parameter files (e.g., `dopdis` makefile and *evalb* settings).

dop/mysql Contains the `mysql` databases with the temporary fragment corpora used in DOP* training if `mysql` is used. Again, if disk space on this file system is scarce, `dop/mysql` can be anchored to another file system.

dop/doc Documentation of the programs.

dop/examples A few example scripts.

It is important that this directory structure be left intact since, e.g., bash scripts in `dop/scripts` locate the `dop/javastuff` directory by:

```
`dirname $0`/../../javastuff
```

The binaries in `dop/bin` have been compiled with Redhat Linux 9.0. For other platforms, the sources should be recompiled by changing into directory `dop/source` and running `make` and then copying all resulting object files (`*.obj`) to `dop/objects`.

The *evalb* program has to be compiled separately: Change into `dop/source_evalb` and type:

```
make
```

2.2 Preparing the environment

First we need to prepare the Linux environment so that the `dopdis` training modules work properly. Conduct the following steps in turn:

- Add `dop/bin/` to your `$PATH` environment variable so that the training programs can be found later when you need them (you might need to `source` your `.cshrc` file and then `rehash` to get things right).

²This is achieved using `"ln -s"`.

- Add `dop/scripts` also to your `$PATH` env variable.
- Create a directory called `D-OBJECTS` directly under your home directory³ and then copy to it all files from `dop/objects`.
- Create a directory, anywhere you like, and put there your treebank (read section 4 for treebank format issues). We will refer to this directory with `MyDIR`.
- To create a `dopdis` parser for a new treebank, you always need the following three files to be available in the directory where your treebank is found (i.e. `MyDIR`): `Make_DOPDIS`, `Train_SingleExperiment.sc` and `parameters.c`. Copy these files from `dop/objects` or from `$HOME/D-OBJECTS` (if you created one⁴). Here is a short explanation of what these files contain:
 - `Train_SingleExperiment.sc` is the training shell script⁵ for extracting a stochastic grammar from the treebank, compiling and linking it with the object code that implements the parsing algorithm (the objects files in `$HOME/D-OBJECTS`).
 - `Make_DOPDIS` is called by `Train_SingleExperiment.sc` for compiling the C code that was generated by `Train_SingleExperiment.sc` and linking it with the object code found in `/dop/objects/` (or `$HOME/D-OBJECTS/` if you created that earlier).
 - `parameters.c` is a C code file containing some parameters for pruning the parse space. These parameters can be modified (see A.2 for more on this).

Now you are set to training a `dopdis` instance on your treebank. We will show how this works with a running example through the following subsections. Now you should be ready to use the **dopdis** programs.

3 Getting Started: An Example

In the following, we will guide you step-by-step through the splitting of a small treebank, the DOP1 training, DOP* training, and the testing of the parsers.

³If you wish to use another directory for this, you will need to change the first line in `Make_DOPDIS` accordingly.

⁴The notation `$HOME` or `${HOME}` stands for the home directory of the user.

⁵For the sake of a simpler exposition we will delay discussion of the training arguments until subsection 5.

3.1 Working with DOP1

The `dopdis` environment contains programs for three aspects of parsing research: (1) Training: extraction of a stochastic grammar from a given treebank and compiling it into a parser, (2) Parsing: the parser `dopdis` can be executed on input sentences, and (3) Evaluation: various programs can be used for PARSEVAL style evaluations of the parser output.

We will go through these steps now. We start first with splitting a given corpus into training and testing parts.

3.1.1 Splitting the corpus

At first, we need to randomly split our treebank into training and testing part. In `dop/examples`, you find a small part of the OVIS corpus (Scha et al., 1996) (file `ovis.small`). Change into the directory `dop/examples/dop_one` and type:

```
viewdoptree <../ovis.small
```

to inspect the trees in that corpus. You can navigate with ‘n’ (next tree) and ‘p’ (previous tree). With ‘P’, you can create an encapsulated postscript file of a tree. Type ‘q’ to quit the tree viewer.

The treebank contains 1000 trees. Let’s randomly split off 30 of them and reserve them for testing:

```
splitcorpus ../ovis.small 30 trainingpart testingpart
```

3.1.2 DOP1 Training

Now we are ready for DOP1 training. It is important to keep in mind that `dopdis` training differs slightly from the training of other parsers: rather than putting the stochastic grammar into tables that are read-in during run-time, in `dopdis` the stochastic grammar is extracted from the treebank and translated into *C code data structures*, which are compiled and linked with the parsing algorithm (also C code). On the one hand this means that training might be slow due to the transformation of the stochastic grammar into optimal C code. On the other hand, this means that loading the parser into memory during parsing should be very fast.

As mentioned earlier, you always need the following three files to be available in the directory where your treebank is: `Make_DOPDIS`, `./Train_SingleExperiment.sc` and `parameters.c`. You can copy these files from `dop/objects` to the directory where you are working.

In the directory `dop/examples/dop_one` the three files are available and so the training command (a script – see Figure 1) can be executed:

```

#!/bin/tcsh

## Training script for the parser:
## takes training corpus file as command line parameter
##

## get rid of words leaving unknown only,
## then get rid of unary productions
## transform into short form: necessary for gen-t-grams
#
pro2shf $1 > .STSG.shf

## Extract the rules into file .STSG
## Parameter: -d<num> where <num> is subtree depth
## (for extracting a PCFG you should change to -d1)
gen-t-grams -p -d3 -i .STSG.shf -o .STSG #>& .DEBUG
##-----
## Transform rules to Extended Chomsky Normal Form (ECNF)
SubCFG2ECNF .STSG > .STSG.ecnf
#
## Transform grammar into a convenient representation
SUB-2-RBSTSG .STSG.ecnf .STSG.rb #>& /dev/null
##
## Transform grammar into C data-structures + binaries
RB-2-CandF -i .STSG.rb -o GrammarSingle.c ; FindChs
##
## clean up earlier stuff for correct compilation
rm -f dopdis TheGramInitSingle.o unary_bamboo_functions.o
## compile and link grammar with parser code
make dopdis -f Make_DOPDIS >.DEBUGmake

## The resulting parser:
## Executable called: dopdis
## Necessary tables in: .CodesList.bin and .ChildPlace.bin
##

```

Figure 1: The (slightly edited) contents of `Train_SingleExperiment.sc`

```
./Train_SingleExperiment.sc trainingpart
```

This should only take a few seconds and create the executable file `dopdis` in the current directory. Run without command line arguments, this parser expects a sentence as standard input and outputs its most probable derivation (in form of a parse tree, in which nodes where substitutions occurred are prefixed by a '*'). Type

```
./dopdis
```

and then as input sentence:

```
nee hoor.
```

If you haven't had an unlucky training/test split, `dopdis` should output the following derivation:

```
(*sss, [( *tvxA, [( tvxA, [( nee_, [] ) ] ], (int, [( hoor_, [] ) ] ) ] ) ] ) .
```

Run `viewdoptree`, paste that derivation into the expected standard input, and press `<Ctrl>-d` (to signal the end of input) in order to view the corresponding tree graphically.

3.1.3 Running the parser

Execute

```
dopdis -h
```

for more information on the workings of `dopdis`.

You may want to use `dopdis` directly within a PARSEVAL experiment: in that case you can skip this subsection and move on to subsection 3.1.4 which embeds the use of `dopdis` the parser into a script that evaluates the output against a gold standard test treebank file.

In principle, `dopdis` runs on a single input sentence at a time. To run `dopdis` on a sequence of sentences, you may use the `map` function available in `dop/bin` as follows:

```
cat file | map "dopdis"
```

where `file` is a file containing a sequence of sentences, *each sentence on a separate line and each ending with a dot* (.). The function `map` applies the parser `dopdis` found in this directory to each sentence in turn and outputs for each sentence the most probable derivation.

It is possible to apply `dopdis` together with various arguments to a sequence of sentences found in a file. For example

```
cat file | map "dopdis -P"
```

will apply a pruned version of `dopdis` to every sentence in the file `file`.

For evaluation purposes against a test treebank it is necessary to apply a detransform the Chomsky Normal Form output of the parser using `RevECNF`):

```
cat file | map "dopdis" | RevECNF
```

In fact, for every transform applied to the training trees, a suitable detransform should be applied to the output of the parser in order to be able to compare. An less preferred alternative is to transform also the test set parses using the same transform: the risk with some kind of transforms (e.g. that add internal nodes to the trees) is that the evaluation results of different transforms become incomparable!!

3.1.4 Evaluating the parser

Now let's test our parser on the 30 reserved testing trees, where we want to evaluate the parses determined by the most probable derivations ('mpd'):

```
testdopdis mytest mpd testingpart .
```

(for DOP models, one could also select "mpp", for most probable parse instead of "mpd"). Note the "." as last argument for `testdopdis` (current directory is the place where the parser is to be found).

Since the output is very redundant (in order to give as many cues as possible in case of errors), the important output is logged in file `./mytest/test_log.txt`.

3.2 Working with DOP*

DOP* is an improved parsing approach for the DOP model. It differs in the way in which the fragment weights are determined. The parser has been implemented in Java but makes use of `dopdis` C programs internally. The program is by default set up to do corpus splitting, training, and testing all in one go, so all we need to do to get going is to change into the directory `dop/examples` and type (beware, this command will take about ten minutes):

```
dopstar myexperiments ovis.small 50 -d3 2
```

This creates a directory `./myexperiments` in which the training and test corpora as well as the compiled parsers and the log file will be stored. A test corpus of 50 trees will be split off the treebank `ovis.small`. The argument '-d3' tells the training program to extract fragments up to depth 3 from the training corpus. The last argument '2' means two experiments (*i.e.*, two different training/test splits of the treebank) will be performed, and during testing, individual results as well as the average results from those two experiments are reported. To see the results, look into the log file `./myexperiments/log_dopstar.txt`.

3.3 Tips

If other people are also using the machine on which you are training `dopdis`, you might want to run your jobs with low priority. You can use the unix command `renice` for that. Type

```
renice 19 -u USER
```

(where USER is your user name) in order to run all subsequent commands / programs started from the same terminal with lowest priority.

When you are remotely logged in for **dopdis** training, a loss of connection automatically kills the running training job on the machine. To avoid that, type:

```
screen
```

This starts a persistent session that will only end when you type the `exit` command. Now you can start the training script and disconnect your remote session. The `screen` session will internally keep going. To reconnect to the screen session, open a terminal, log into the machine and type:

```
screen -r
```

3.4 What's next?

In the next sections we will discuss some of the details of the training and using **dopdis**. We will discuss the treebank format in section 4, the training process in section 5 including how to specify the kind of probabilistic grammar to be extracted from the training treebank, a version for improved estimation of probabilities in section 6, and various other information that can be useful for using **dopdis**.

4 Treebank Format and Viewing

Treebanks employed by **dopdis** are text files each of whose line represents a tree in the following format:

```
tree → (root, [childtrees]).
childtrees → childtree, childtrees
childtrees → childtree
childtree → (category, [childtrees])
childtree → (word_, [])
```

To view a treebank graphically, use the script `viewdoptree`, which utilizes the `viewtree` wish script by Adwait Ratnaparkhi. Example:

```
viewdoptree <dop/examples/ovis.small
```

Symbols: Unlike nonterminal symbols, terminal symbols always end with an underscore symbol `"_"` (the underscore should not appear anywhere else except at the end). The character `"@"` is also reserved and is used at the end of nonterminal symbols to symbolize nodes that are added during a transform into Chomsky Normal Form (CNF) (these symbols are extracted out of the tree using the detransformation `Rev-ECNF`).

The nonterminal symbols `"ssss"`, `"sss"`, `"xxxprior"`, `"xxxunknown"` and `"xxxphrase"` are reserved for internal use of the parser.

The format of the symbols is as follows (using `flex` format): the last two lines provide the definition using the “—” that stands for alternative definitions (formally, the union of different languages):

```
ecnfsym ("@" )
dollar (" $" "_" )
numword ([a-zA-Z0-9\ -][0-9a-zA-Z\ -]* "_" )
number ([0-9]+ "_" )
real ([0-9]*"."[0-9]* "_" )
hour ([0-9]+":"[0-9]+ "_" )
word ([a-zA-Z\ '\ -][a-zA-Z\ '\ -]* "_" )
time ([a-zA-Z]"."[a-zA-Z][\ .]* "_" )
dreals ([0-9][0-9]*"."[0-9][0-9]*)
emreals ([0-9][0-9]*"."[0-9][0-9]*"e-"[0-9][0-9]*)
epreals ([0-9][0-9]*"."[0-9][0-9]*"e+"[0-9][0-9]*)
reals {dreals}|{emreals}|{epreals}

nonterminalsX ([a-zA-Z]+[\^\%\-a-zA-Z0-9\@]*{ecnfsym})
nonterminalsY ([a-zA-Z]+[\^\%\-a-zA-Z0-9\@]*)

terminals {dollar}|{numword}|{number}|{hour}|{time}|{word}|{real}
nonterminals {nonterminalsX}|{nonterminalsY}
```

5 Training `dopdis`: Basic Programs

When working with `treebank` grammars, including `DOP`, the important bit is in specifying what probabilistic grammar to extract from the training `treebank`. The shell script

`Train_SingleExperiment.sc` mentioned earlier embeds the extraction process within a complete training process (including all kinds of transforms on the extracted grammar that optimize the resulting parser). The program which extracts the grammar from a `treebank` is called

`gen-t-grams`, and is the only program for which the user might need to change the arguments (inside the file `Train_SingleExperiment.s`).

The following list gives an overview of the basic `dopdis` training programs. For an example usage of each of the programs, see the file:

```
dop/examples/dop_one/Train_SingleExperiment.sc
```

To obtain more information about a program `prog`, just type:

```
prog -h
```

Some extra information is included in the appendix A.

pro2shf Takes a `treebank` as input (file name can be passed on the command line).

Prepares this `treebank` for `gen-t-grams`. Eliminates unary productions. The resulting file is passed as standard output.

gen-t-grams Takes a preprocessed treebank file as input. Extracts rules (i.e., DOP subtrees or fragments) and outputs them line by line. The type of rules to be extracted can be restrained (e.g., extracting only fragments up to depth three). Run `gen-t-grams -h` for details of the arguments or see Appendix A.2. The most important arguments are: `-p` (for extracting priors necessary for pruning) and `-d<num>`, where `<num>` is an integer greater than zero which specifies the maximal depth of an allowed subtree/fragment. Hence, for extracting a Probabilistic Context-Free Grammar (PCFG) one should use `gen-t-grams -p -d1`. It is recommended always to use the argument `-p` in order to obtain a parser that is capable of pruning.

SubCFG2ECNF Transforms the input rules/subtrees into Extended Chomsky Normal Form (ECNF). Note that after parsing a de-transform (called `Rev_ECNF`) should be applied to the trees output by the parser. This is done implicitly in the testing and evaluation environment called `testdopdis`.

SUB-2-RBSTSG Transforms a given ECNF grammar into an intermediate representation which is easier to transform into C code. The resulting file is passed as standard output and a C file `.unary_file.c` is created.

RB-2-CandF Transforms a binary representation returned by `SUB-2-RBSTSG` into C data-structures and extra binary files. The standard output (the C file) should be redirected into `GrammarSingle.c`. A binary `.CodesList.bin` will be created.

FindChs Conducts some efficiency compilations. The binary file `.ChildPlace.bin` will be created.

dopdis The executable parser that is created after the above programs have been executed. To create it, run the command:

```
make dopdis -f Make_DOPDIS. The parser consists of the the three file
dopdis, .CodesList.bin and .ChildPlace.bin which should be in
the same directory for the parser to work properly.
```

If run without any options, the parser expects the sentence to be parsed as a space-separated list of words followed by a dot in the standard input and outputs the most probable derivation (mpd). Type `dopdis -h` for more information.

6 The DOP* estimation variant

DOP* (Zollmann, 2004) is a DOP variant that assigns weights to fragments from one part of the training corpus in such a way that the likelihood of the parses in the other part of the training corpus is maximized. This estimation method leads to an estimation-theoretically consistent parser, which improves over the original (DOP1)

estimator in parsing accuracy and reduces parsing time exponentially. Also the training time is considerably reduced if fragments of great depth are to be extracted from the training corpus.

6.1 Invocation

DOP* is run via the script `dopstar`. Typing ‘`dopstar`’ without arguments displays a small help screen.

6.1.1 Training mode

To invoke DOP* for training, the usage is:

```
dopstar TARGET_DIR CORPUS_FILE TESTING_CORPUS_SIZE
      FRAGMENT_OPTIONS N_O_EXPERIMENTS [START_WITH_EXPERIMENT]
```

The arguments are as follows:

TARGET_DIR Root of the directory structure with the training and testing files. Will be created if necessary.

CORPUS_FILE Treebank file, which will be split into training and testing corpus.

TESTING_CORPUS_SIZE Size of the testing corpus to be randomly split off from **CORPUS_FILE**.

FRAGMENT_OPTIONS The options for the fragment extraction, which will be passed on to gen-t-grams. If white-space is used, the options must be passed in ‘`literals`’.

N_O_EXPERIMENTS Number of experiments.

START_WITH_EXPERIMENT Number of the experiment to start with in case some experiments are to be left out.

The following is an example of an invocation for training purposes:

```
dopstar /myexperiments examples/ovis.complete 1000 '-d3 -n2' 5
```

After all training experiments have been done, the program automatically switches into testing mode for the most probable parse. If you don’t want to test, you can just interrupt the program by pressing `<Ctrl>-d`.

6.1.2 Testing mode

To invoke DOP* for evaluation of previously trained parsers, the usage is:

```
dopstar TARGET_DIR TEST N_O_EXPERIMENTS [START_WITH_EXPERIMENT]
```

The arguments `TARGET_DIR`, `N_O_EXPERIMENTS`, and `START_WITH_EXPERIMENT` are as given above. `TEST` is one of the values `testmpd`, `testmpp`, and `testmrp` and determines whether the parsers are being evaluated according to most probable derivation, most prob. parse, or maximum recall parse, respectively.

Here is an example for MPD-testing of previously trained experiments 3, 4, and 5:

```
dopstar /myexperiments testmpd 5 3
```

6.2 Settings

DOP* has quite a few default settings, which all can be adjusted by altering the corresponding static variables of the main class `DOPStar` located in file `dop/javastuff/DOPStar.java`.

6.2.1 Corpus type

By default, DOP* is set up to work with the OVIS corpus (static boolean `ovis` of class `DOPStar` equals `true`). The OVIS corpus (Scha et al., 1996) is somewhat special in the sense that during the training process, the *lexicon*, *i.e.*, each CFG rule of the form

$$\begin{array}{c} N \\ | \\ w \end{array} ,$$

where N is a part-of-speech tag and w is a word, is extracted from the complete corpus (that is, training *and* testing part). These rules are then incorporated into the fragment corpus with (in theory) infinitesimal weights. In standard corpora, where no lexicon is to be extracted during training, the variable `DOPStar.ovis` should be set to `false`.

6.2.2 Using `mysql` for DOP* training

By default, the public static boolean variable `sqlMode` in class `FragmentCorpus` is set to `false`. The DOP* training process can be sped up considerably when `mysql` is used, an open-source SQL server available at <http://dev.mysql.com/>. A sample setup⁶ and a start⁷ script are contained in `/dop/mysql`.

To have DOP* connect to the `mysql` server for storing the fragment corpora temporarily created during training, set `sqlMode` to `true`.

⁶To be used after installing `mysql`.

⁷To be used whenever the machine has rebooted, running it multiple times does not matter, so it can also be run each time DOP* is being executed.

7 Setting the pruning parameters for `dopdis`

Go to the directory where you are making a `dopdis` instance. You should find the file `parameters.c` there. Open this file with some text editor. Change the parameters as described next, save the file and recompile `dopdis` (`make -f Make_DOPDIS` will incorporate your changes in the `dopdis` instance you are building).

7.1 Kind of pruning and limitations

The pruning method in `dopdis` (version 0.1) is a simple combination of beam-width based on two parameters: number of edges allowed in an entry and a threshold probability relative to the maximum probability edge in the entry. In the current version there is no stack for eliminated nodes. Hence, the parser does not backtrack if no parse is found for the input sentence.

Furthermore, the pruning parameters are set manually according to user experience. A better way is to incorporate an automatic method for setting such parameters based on parser coverage of a held-out set.

These limitations might be dealt with in future versions of `dopdis`.

7.2 Parameter details

A note: The C language type `ProbDomain` specifies that a certain real number is of type `ProbDomain` (probability domain – double) and makes it compatible with the rest of the definitions of probabilities within the parser.

The file `parameter.c` is in the directory where the `dopdis` parser is being built. This file contains the pruning parameters. At the moment these parameters are set by hand. You can also copy this file from the directory `dop/objs`.

There are only two places in the file `parameters.c` that should be changed. These concern the first few lines only. The rest of the file `parameters.c` should be left as is. Do not change any part not mentioned below since this might affect compilation and correctness!

Beam width: The first line fixes the beam width:

```
#define _BEAM_WIDTH ((_sen_length > 20) ? 60 :  
                    ((_sen_length > 10) ? 65 : 100) )
```

The beam width is the maximal number of nodes in every chart entry that may remain after pruning. The top `BEAM_WIDTH` ranking nodes are retained, while the rest is discarded with.

There is a possibility for defining different BEAM_WIDTH values for different sentence lengths: for example, below you see that sentences of length larger than 20 have BEAM_WIDTH=60, of length between 10 and 20 BEAM_WIDTH=65 and otherwise of BEAM_WIDTH=100.

The following line is currently not in use:

```
#define _BEAM_WIDTH_SMALL_SPAN(span) (( _sen_length > 10) ? 5 : 4) )
```

Probability threshold: Another set of parameters are found in the lines concerning further pruning by a threshold: these concern the ratio of the probability of some node that is being pruned to the node with maximum probability in the same chart entry. The ratio is expressed as LOG(10): hence 2.0 means 100 times (i.e. the current node has probability that is more than 100 time smaller than the most probable node in the same entry). The most important definition concerns nodes that are roots of DOP subtrees (or left-hand sides of PCFG rules):

```
#define _ROOT_P_RATIO_FST \
(( _sen_length > 60) ? ((ProbDomain) 2.10) :
(( _sen_length > 45) ? ((ProbDomain) 2.40) :
(( _sen_length > 35) ? ((ProbDomain) 3.0) :
(( _sen_length > 25) ? ((ProbDomain) 3.5) :
(( _sen_length > 20) ? (ProbDomain) 3.8 : (ProbDomain) 4.0)) )) )
```

Less crucial is the following definition concerning nodes that are internal (non-root) to DOP subtrees (one may simply state very high numbers here e.g. 10 or so).

```
#define _INTERN_P_RATIO_FST \
( ( _sen_length > 60) ? ((ProbDomain) 10.0) :
( ( _sen_length > 45) ? ((ProbDomain) 10.60) :
( ( _sen_length > 35) ? ((ProbDomain) 10.2) :
( ( _sen_length > 25) ? ((ProbDomain) 10.7) :
( ( _sen_length > 20) ? (ProbDomain) 10.0 :
(ProbDomain) 10.0)) )) ) )
```

The rest of the file `parameters.c` is should be left in tact. Do not change that as this might affect compilation and correctness.

8 Further Documentation

For more information concerning DOP see (Bod et al., 2003), and concerning the OVIS corpus see (Scha et al., 1996). For more information concerning `dopdis` environment contact Khalil Sima'an (simaan@science.uva.nl).

9 Credits

The `dopdis` environment (training programs and parsing algorithms) was written by Khalil Sima'an. Andreas Zollmann wrote the DOP* estimation programs, and various user friendly scripts that integrate training and testing modules from the original environment.

10 Acknowledgments

A few Perl scripts in the `dopdis` environment originate from scripts written by Remko Bonnema. Luciano Buratto provided the python scripts used by DOP*. The persistent hash table used by DOP* whenever `mysql` is not available is part of the package `jdbm.jar`, which is open-source and maintained at <http://jdbm.sourceforge.net/>.

The open-source *evalb* program used for parsing evaluation is due to Satoshi Sekine and Mike Collins.

References

- R. Bod and R. Scha. 1996. *Data-Oriented Language Processing: An Overview*. Research report nr. LP-96-13, ILLC Research reports, University of Amsterdam, www.essex.ac.uk/linguistics/clmt/papers/dop/bodscha.ps.
- R. Bod, R. Scha, and K. Sima'an, editors. 2003. *Data Oriented Parsing*. CSLI Publications, Stanford University, Stanford, California, USA. not refereed.
- R. Scha, R. Bonnema, R. Bod, and K. Sima'an. 1996. *Disambiguation and Interpretation of Wordgraphs using Data Oriented Parsing*. Technical Report #31, NWO, Priority Programme Language and Speech Technology, <http://grid.let.rug.nl:4321/>.
- K. Sima'an. 1999. *Learning Efficient Disambiguation*. PhD dissertation (University of Utrecht). ILLC dissertation series 1999-02, University of Amsterdam, Amsterdam. Available at <http://staff.science.uva.nl/~simaan/D-Papers/MyThesis.ps>.
- A. Zollmann. 2004. A Consistent and Efficient Estimator for the Data-Oriented Parsing Model. Master's thesis, Institute for Logic, Language and Computation, University of Amsterdam, Netherlands, May. Available at <http://staff.science.uva.nl/~azollman/publications.html>.

A Details on major `dopdis` programs

A.1 Parameters of `gen-t-grams`

A major program in the `dopdis` environment is `gen-t-grams`. This program extracts a stochastic DOP grammar (Stochastic Tree-Substitution Grammar – STSG) according to the arguments specified for this program. Trivially, since PCFGs are subsumed by STSGs, one can use this program also to extract PCFGs from the treebank.

```
Usage: gen-t-grams [options] -i <infile> -o <outfile>
Usage: gen-t-grams [options] <infile> [outfile]
```

NOTES on input:

The string `xxxempty_` denotes an epsilon !!!
Epsilons do not count as words. Subtrees that are fully "lexicalized" only by epsilons have depth zero

NOTES on output:

Besides the desired subtrees and their probabilities, the program can also output (model independent) prior probabilities for non-terminals in the form of subtrees:
(XP,[tree(xxxprior,[])], PRP(XP)),
where $PRP(XP) = (freq(XP) / TFreq)$; $TFreq$ is the total number of occurrences of all non-terminals in the tree-bank.

The options are:

Major options:

```
-d<num> maximal depth of subtrees (default 1)
-p generate also subtrees that represent the prior
  probabilities of non-terminals
-n<num> max number of open nodes (default 200)
-l<num> max number of lexical items (default 200)
-L<num> max number of consecutive lexical items (default 200)
-w<num> min number of lexical items in subtrees of
  depth > 1 (default 0)
-f<num> a threshold (integer) on the frequency of
  a subtree (default 0)
-&<num> a threshold (integer) on the frequency of
  a node in a subtree (default 0)
-C[w|o|h] select corpus:
  w: for WSJ (Wall Street Journal)
  o: for OVIS (OpenbaarVervoer Informatie Systeem)
  h: for Hebrew corpus
```

Minor options:

```
-q Run quietly
-l Generate all subtrees of depth 1
-b<num> maximum branching factor of a node (default 200)
-I<num> maximum branching factor of every INTERNAL node
  except root (default 200)
```

- m<num> maximum branching factor of every node labeled with a modifier T-gram (default 200)
- T always allow lexicalization of POS-TAGS (default is false)
- H some pre-specified POS-TAGS not to be lexicalized in subtrees (default false)
- K Subtrees must be fully lexicalized ONLY at their left corner
- k Subtrees must be fully lexicalized at their left corner (possibly elsewhere also)
- Z[y|n] y: Generate only lexicon subtrees
n: do not generate lexicon subtrees at all
- U Unary rules that do not involve POSTAGs are internal to subtrees
- x No frequency or probabilities in output subtrees
- v Subtree probabilities as a single distribution
- F Compute frequencies rather than probabilities for subtrees
- E Generate also rules to allow the probabilistic evaluation of sequences of symbols possibly containing non-terminals.

A.2 The arguments of dopdis

Note that dopdis runs on a single input sentence at a time. To run dopdis on a sequence of sentences, you may use the map function available in dop/bin as follows:

```
cat file | map "dopdis"
```

where file is a file of sentences *each on a separate line and each ending with a dot* (.). The parser will run at each sentence in turn and out for each sentence the most probable derivation/parse. Note that the output is in Extended Chomsky Normal Form and needs to be de-transformed before evaluation: use RevECNF for this as follows:

```
cat file | map "dopdis" | RevECNF
```

When you have a dopdis instance, you may specify arguments for running it on a single input sentence. To view these arguments run `dopdis -h`. You should get the following:

```
Usage: dopdis [options] -i <infile> -o <outfile>
or: dopdis [options][infile] [outfile]
```

Information options:

- I About this parser (information)
- 0 Print the input format that this parser expects
- h Print this message

Tree-bank options (for some morphology of unknown

words - only when parser is trained in a suitable fashion!!):

- T [WNHG]:
WSJ (W) / Negra (N) / Hebrew (H) / General (G) (default)

First options:

- T[]

-l<int> sentences longer than <int> are discarded
 -U Do NOT attempt resolving unknown words
 -@ Do not try to avoid cycles (for OVIS)
 -!<directory> specifies that the binary files .CodesFile.bin and
 .ChildsPaces.bin are found at directory
 -f[0,1] Filter according to constituents given in the input
 (CAT/word style)
 0 implies given CAT is Prefix of allowed CAT
 1 implies given CAT is Equal to allowed CAT
 -# Parsing PoSTag sequences instead of morpheme sequences (for Hebrew)

Options for disambiguation algorithm (instead of MPD):

-m<#num> Select the Most Probable Parse (MPP) (default is MPD)
 using <#num> sample-size
 -M<#num> like -m but also printout all sampled parses
 -K<#num> like -m but also printout all sampled derivations
 -d Compute Goodman's Max Recall Rate Parse
 -B Compute both Goodman's Max Recall Rate Parse and the MPD
 -A<#fac> Interpolate Max Recall Rate with Context-Free Recall Rate
 and Parent-Child Rate Linear interpolation
 $fac * LRR + ((1-fac)/2) * CFPR + ((1-fac)/2) * PaCh$

More options:

-+<#fac> Interpolate Wordgraph probs (fac) with
 Lexical Grammar Probabilities (1-fac)
 -C Do all chains of unaries (see default below -
 but break any chains deeper than 20)
 -P Prune derivation-forest using BEAM-with-Priors followed by
 Global-Thresholding: implies Semi-interleaved !!
 -S Semi-Interleaved computation of parse- and derivation-forest
 -b Apply a bigram-language model for some pruning of the
 Part-Of-Speech Tags before parsing
 -t Print-out the Part-Of-Speech Tags that are assigned to
 the words (don't parse)
 -N Only recognize, don't disambiguate, don't build parse forest
 -R Parse under the assumption that the grammar is a
 right linear grammar (finite-state)
 -W Build WHOLE derivation-forest (don't optimize assuming MPD)
 -s spare memory in pruning (simply does stricter pruning)
 -H (Hebrew Only: POSTAG in input) Match unknown POSTAGs in input
 to known ones by similarity
 [DEFAULT: all options are set OFF]

NOTES: 1. The symbols "sss", "ssss", "xxxprior" and "xxxunknown" are
 reserved symbols.
 2. This parser might have problems with cyclic rules X->X
 3. This parser cuts off chains of unary rules at chain-length 2.
 4. This parser does not accept epsilon rules.

A.3 The input format for `dopdis`

The parser `dopdis` (a trained and compiled parser) has a wide range of formats, a simple input sentence, a stochastic finite state automaton (such as a word-graph output by a speech recognizer) or a set of labeled constituents with probabilities for the terminals (words).

This parser has various input formats:

- A. Simple sentence as sequence of words (form I below)
- B. Stochastic Finite State Automaton (form II below)
This allows the output of e.g. a speech-recognizer to be pipelined to the parser
- C. A set of labeled constituents (form III below)
A set of constituents in the input is used to filter (option `-f[0,1]`) the parse-space only to those parses that include the given constituents

Definitions: -Words or terminals (`[a-zA-Z]+_`)
-Nonterminals (`[a-zA-Z][a-zA-Z-]*`)
(`[a-zA-Z][a-zA-Z-]*@`)
-Reserved are the words:
XXXunknown, xxunknown, xxxprior, xxxphrase_
-Below $1 \leq i < j \leq (\text{sentence_length}+1)$

Format I: Sequence of words followed with a full stop "."

Format II: Sequence of transitions followed with a full stop "." where:
A transition is a quadruple "i j word l_prob"
where $(1 \leq i < j \leq \text{sentence_length}+1)$;
l_prob is $\log_{10}(\text{probability})$
l_prob is optional.

Format III: Sequence of labeled-constituents followed with a full stop "." where:
A labeled-constituent is a five-tuple
"i j CAT/word probability"
(probability may be omitted where
 $(1 \leq i < j \leq \text{sentence_length}+1)$;
probability is $\log_{10}(\text{probability})$;
CAT is a phrasal category (non-terminal)
such that if the category CAT is not a postag,
i.e. a higher level category, then $(j-i > 1)$
and $(\text{word} = \text{xxxphrase}_)$).