The Wall Street Journal experiment (and useful programs)

Antoine Rozenknop

[1]Technical report ID: IC/2004/32

March 26, 2004

[1]http://ic2.epfl.ch/publications/documents/IC_TECH_REPORT_200432.pdf

Abstract

This document gives information on parsing experiments applied to the standard Wall Street Journal corpus ("Standard" means that this corpus has been widely used for exhibiting parsing tests of various models). The tested syntactic models are : standard Stochastic Context-Free Grammars, standard Tree Substitution Grammars, Gibbsian Context-Free Grammars and Gibbsian Tree Substitution Grammars. The parsing experiments are described with deep details so as to enable reader to easily redo the experiments from scratch (i.e. preparing the database, training and evaluating the models). The programs developped for these experiments are also described.

The first chapter describes "from scratch"[1] the experiments and gives the results obtained up to this day.

The second chapter is for people that would want to redo the experiments, or similar ones : it can be read in parallel with the first one and describes the programs and the command lines used for these experiments.

The appendix is a list of available programs, that can be useful for viewing treebanks, computing grammar models, evaluating results. It describes the majority of the programs I used during my PhD.

This document does not contain any explanation about models, theories or algorithms. The experiments it describes could be better understood after the reading of my PhD document : they are the same as the ones presented in that document; only the corpus has changed.

[1] starting from the Wall Street Journal corpus CDROM

Contents

Chapter 1

Making of the treebank

In order to deal with the Wall Street Journal treebank, we have to put it in a suitable form.

1.1 Some observations

The corpus on the CD is a set of files, each file containing one or several parses. A parse looks like this :

```
( (S
    (NP-SBJ
      (NP (NNP Pierre) (NNP Vinken) )
      (, ,)
      (ADJP
        (NP (CD 61) (NNS years) )
        (JJ old) )
      (, ,) )
    (VP (MD will)
      (VP (VB join)
        (NP (DT the) (NN board) )
        (PP-CLR (IN as)
          (NP (DT a) (JJ nonexecutive) (NN director) ))
        (NP-TMP (NNP Nov.) (CD 29) )))
    (. .) ))
```

As can be seen on this example :

- parses are dispatched over several lines;
- parentheses "(" and ")" serve as delimiters
- the labels of groups can be either one symbol (e.g. NP) or several symbols (e.g. PP-CLR, which is the concatenation of PP and CLR). In the case where several symbols are concatenated, the first one is the syntactic category of the group, the second one is its semantic function.

- some labels contain a number as “S-TPC-1” in the following parse :

```
( (S
    (S-TPC-1
      (NP-SBJ
        (NP
          (NP (DT A) (NN form) )
          (PP (IN of)
            (NP (NN asbestos) )))
        (RRC
          (ADVP-TMP (RB once) )
          (VP (VBN used)
            (NP (-NONE- *) )
            (S-CLR
              (NP-SBJ (-NONE- *) )
              (VP (TO to)
                (VP (VB make)
                  (NP (NNP Kent) (NN cigarette) (NNS filters) ))))))))
      (VP (VBZ has)
        (VP (VBN caused)
          (NP
            (NP (DT a) (JJ high) (NN percentage) )
            (PP (IN of)
              (NP (NN cancer) (NNS deaths) ))
            (PP-LOC (IN among)
              (NP
                (NP (DT a) (NN group) )
                (PP (IN of)
                  (NP
                    (NP (NNS workers) )
                    (RRC
                      (VP (VBN exposed)
                        (NP (-NONE- *) )
                        (PP-CLR (TO to)
                          (NP (PRP it) ))
                        (ADVP-TMP
                          (NP
                            (QP (RBR more) (IN than) (CD 30) )
                            (NNS years) )
                          (IN ago) ))))))))))
    (, ,)
    (NP-SBJ (NNS researchers) )
    (VP (VBD reported)
      (SBAR (-NONE- 0)
        (S (-NONE- *T*-1) )))
```

```
(. .) ))
```

This numeric symbol denotes the reference of the group. This reference can be used in traces : a trace is denoted by "-NONE-" symbol, followed by "*T*-1", or *T*-2"... where the numeric symbol is used to indicate the referenced group. The "-NONE-" symbol can also be followed by "*", which means that there is no referenced group : it then only denotes the place of a "ghost" group with no surfacic word.

1.2 Translating the treebank

(see command detail in section 2.1)

Several steps are necessary in order to obtain a treebank that can be used with our tools :

1.2.1 One file

(see command detail in section 2.1.1)
The parses have been collected in one single file.

1.2.2 Removing the traces

(see command detail in section 2.1.2)
The traces have been removed :

- the numbers in the syntactic labels have been removed;
- the "-NONE-" symbols have been removed, as well as every symbol under it, and every symbol that dominated uniquely the branch containing the "-NONE-" symbol.

1.2.3 Removing the quotation marks

(see command detail in section 2.1.2)
In order to obtain the treebank the closest to the one used by other researchers (Bod), the quotation marks – " and " – have been removed from the terminal symbols, as well as the symbols that uniquely dominate them.

1.2.4 Removing the semantic symbols

(see command detail in section 2.1.2)
The standard PARSEVAL scores do not take into account the semantic tags appearing in some labels. Thus, these semantic tags have be removed from the treebank, so that the each label contains only one syntactic symbol.

1.2.5 Removing the cycles

(see command detail in section 2.1.2)
In some parses, there are symbols that dominate only one single other symbol, with a context-free rule like "$X \longrightarrow Y$". This can lead to cycles in the grammar : if "$Y \longrightarrow X$" is also present in the treeebank, there would be an infinity of parses containing the X symbol ($X \longrightarrow Y \longrightarrow X$, $X \longrightarrow Y \longrightarrow X \longrightarrow Y \longrightarrow X$...). This is called a cycle. Our parser can cope with this phenomenon, but our gibbsian training system cannot. So, all these "non-branched" symbols but the preterminal ones have been removed from the treebank.

Some other methods have been tried to get rid of the cycles in the grammar. It is possible to remove only a subset of the non-branching nodes of the treebank and obtain a grammar with no cycle. Algorithms have been developped for that. But there is only minor difference in the resulting treebank, and the "remove all threads" method is much faster than all others !

1.2.6 Renaming the preterminal symbols

(see command detail in section 2.1.2)

Preterminal symbols are symbols that dominate a terminal (or "surfacic") symbol of the parses, i.e. a leaf of a parse tree, or a "word" of the language.

The parser we use need those symbols to be named ":1", ":2", ":3"... Thus, each preterminal symbol have been replaced in the treebank by a numeric symbol preceeded by ":". A correspondance table is created to keep track of this transformation.

1.2.7 (Optionnal) Removing the terminal symbols

(see command detail in section 2.1.3)
In some experiments, we want to parse a serie of morphosyntactic tags and not a serie of words. For these experiments, we have to remove the words from the treebank. This is done by replacing the words (i.e. the leaves) in the parse trees by the (preterminal) symbols that immediately dominate them. Then, those preterminal symbols become also terminal symbols of the grammar. Terminal rules of the grammar all become ": $1 \longrightarrow$: 1", ": $2 \longrightarrow$: 2"...

1.2.8 Writing the resulting treebank

The resulting treebank is written into a single file, one parse per line; the delimiter of syntactic groups are brackets – "[" and "]".

1.2.9 Splitting the treebank

(see command detail in section 2.1.4)
The standard partition of the treebank is the following :

- sections 2 to 21 of the Wall Street Journal corpus are used to train the models;
- section 22 is (optionnally used) to cross-validate the model;
- section 23 is used to test the model.

For us, this means we have to split our treebank into two files. The first file is the training corpus. It contains 39832 trees, starting from the tree #3915 of our unique file. The second file is the test corpus. It contains 2416 trees, starting from tree #45447 of our unique file.

1.2.10 Extracting the sentences

(see command detail in section 2.1.5)
The sentences (i.e. the leaves of parse trees) are extracted from the treebank test file in order to be parsed in the testing phase of the experiments.

1.3 Making of the models

(see command detail in section 2.2)

With the training corpus, we can build different syntactic models :

1.3.1 SCFG

(see command detail in section 2.2.1)
We build two SCFGs for the experiments. The first one, denoted as "scfgComplete", is computed by extracting all context-free rules from the complete corpus (i.e. test+training). The second one, denoted as "scfg", is computed by extracting all context-free rules from the training corpus. In the parsing tests, we will use the lexicon of "scfgComplete" and the syntactic rules of "scfg". This is to ensure every sentence of the test file can be parsed by our model, as we have not implemented any "guessing" algorithm for finding out the part-of-speech tag for the words.

Statistics of the resulting SCFGs :

Model	Nb. terminals	Nb. POS tags	Nb. syntactic symbols	Nb. rules
Lexicalised SCFGcomplete	52990	43	26	14393
Lexicalised SCFG	51654	43	26	13957
Delexicalised SCFGComplete	43	43	26	15780
Delexicalised SCFG	43	43	26	13957

Notice that the number of rules of "Delexicalised SCFGComplete" is greater than the number of rules of "Lexicalised SCFGcomplete". This is due to the fact that the first one

has been created from the "training + test" treebank (42248 trees), whereas the second one has been created from the complete treebank (training + test + sections 1 and 23 of the WSJ corpus : 49208 trees). This does not change much for the following, because we will never use the syntactic rules of SCFGComplete, but only its lexicon.

1.3.2 GCFG

(see command detail in section 2.2.3)

A Gibbsian Context-Free Grammar can be built for each SCFG, provided that we have access to a suitable training treebank. However, the computation of such a grammar is quite heavy with the "Improved Iterative Scaling" algorithm. The solution found to make this training feasible with the Wall Street Journal corpus was :

- parallelization of the training algorithm : approximately one hundred Sun stations were used;
- restriction of the training corpus to the trees with a maximum of 22 leaves. (see command detail in section 2.2.2)

The resulting grammar has the same rules than the SCFG. However, some of the parameters associated with the rules that do not appear in the "max22-leaves" corpus are set to $-\infty$ by the training algorithm.

The GCFGs have been computed with the lexicons of SCFGcomplete and the CF rules of SCFGs.

Statistics of the resulting SCFGs :

Model	Nb. terminals	Nb. POS tags	NB.syn. symbols	NB rules
Lexicalised GCFG	52990	43	26	13957
Delexicalised SCFG	43	43	26	13957

1.3.3 Head-Driven Stochastic Tree-Substitution Grammar

(see command detail in section 2.2.4)

A Head-Driven STSG has been built for each model, based on Collins' rules to find the head of each syntactic CF rule. Again, the lexicon of the STSGs come from "SCFGcomplete", and the elementary trees of the grammars are computed from the training corpus alone.

Statistics of the resulting STSGs :

Model	Nb. terminals	Nb. POS tags	Nb.syntactic symbols	Nb. elem. trees
Lexicalised STSG	52990	43	26	200536
Delexicalised STSG	43	43	26	52014

1.3.4 Head-Driven Gibbsian Tree-Substitution Grammar

(see command detail in section 2.2.5)
As before, a gibbsian version of the STSGs hav been computed. The complexity problems and their solutions are the same than for the computation of GCFGs : we used the same "max-22-leaves" treebanks than for GCFGs.

1.3.5 Min-Max Stochastic Tree-Substitution Grammar

(see command detail in section 2.2.6)
A Min-Max STSG has been built for each model. Again, the lexicon of the STSGs come from "SCFGcomplete", and the elementary trees of the grammars are computed from the training corpus alone.

Statistics of the resulting STSGs :

Model	Nb. terminals	Nb. POS tags	NB.syn. symbols	NB elem. trees
Lexicalised STSG	52990	43	26	522152
Delexicalised STSG	43	43	26	299483

1.3.6 Min-Max Gibbsian Tree-Substitution Grammar

(see command detail in section 2.2.7)
As before, a gibbsian version of the STSGs have been computed. We used the same "max-22-leaves" treebanks than for GCFGs.

1.3.7 Results of the experiments

(see command detail in section 2.3)

The collected results are shown in the following table.

model	exact	false	false bracketing	crossed bracketing	label precision	label recall
SynLexNothread (Lexicalized treebank without thread)						
scfg	173	2243	2096	1717	0.8344	0.8183
head-driven STSG	39	2377	1936	1521	0.7629	0.7574
head-driven GTSG	54	2362	1918	1549	0.7536	0.7556
min-max STSG	52	2364	2112	1724	0.7615	0.7488
min-max GTSG	untractable computation (needs parallel computation, but too heavy for most machines)					
SynNothread (Delexicalized treebank without thread)						
scfg	268	2148	2067	1712	0.8667	0.8493
gcfg	351	2065	2005	1650	0.8803	0.8671
head-driven STSG	310	2106	2000	1592	0.8745	0.8644
head-driven GTSG	383	2033	1966	1578	0.8800	0.8746
min-max STSG	340	2076	1997	1687	0.8581	0.8534
min-max GTSG	should be tractable, but has not been tested					

Chapter 2

Command lines

This chapter explains in more details the programs and command lines used for buiding and testing the previously exposed models.

2.1 Translating the treebank

(see section 1.2)

2.1.1 One file

(see section 1.2.1)

This is the Command line for merging all trees of the Wall Street Journal Corpus into a single file :

```
find . -name 'wsj*' -exec tgrep -h -e '.*' {} \; > ! wsj00-24
```

2.1.2 Adaptation of the treebank

The transformations applied to the treebank can be achieved with a single run of the python program **traite_wsj.py**.

This program takes two arguments : the source file and the destination file. The source file is in WSJ format (as described in section 1.1), and the output is in SLPToolkit format.

The additionnal transformations to be applied are set by optional parameters.

Removing the traces

(see section 1.2.2)

traite_wsj.py removes syntactic traces from the treebank when it reads the `-notrace` optional parameter on the command line.

Removing the quotation marks

(see section 1.2.3)
traite_wsj.py removes quotation marks from the treebank when it reads the `-noquotes` optional parameter on the command line.

Removing the semantic symbols

(see section 1.2.4)

traite_wsj.py removes semantic symbols from the treebank labels when it reads the `-nosem` optional parameter on the command line.

Removing the cycles

(see section 1.2.5)

traite_wsj.py removes the non-terminal threads from the treebank when it reads the `-nothread` optional parameter on the command line. This will lead to a grammar without cycle.

If the optional parameter `-smartthreads` is passed instead, the program will try to keep as many threads as possible, while insuring that the resulting grammar will be without cycle. The algorithm takes more time than the `-nothread` one.

Renaming the preterminal symbols

(see section 1.2.6)

traite_wsj.py renames the preterminal symbols according to SLPToolkit rules when it reads the `-traitePreterminaux=<filename>` optional parameter on the command line. The `<filename>` argument is the name of the file which the correspondance between original and modified symbols will be written to.

2.1.3 Removing the terminal symbols

(see section 1.2.7)

The program **delexicalise_corpus.py** replaces each terminal symbol in the treebank with the symbol that dominates it. This program takes two parameters : the source file and the destination file. Both files are in SLPToolkit format.

2.1.4 Splitting the treebank

(see section 1.2.9)

The resulting file (named “Arbres”) is simply split into two files
by the following commands :

```
tail +3915  Arbres  | head -39832 > learnArbres
tail +45447 Arbres  | head -2416 > testArbres
```

2.1.5 Extracting the sentences

(see section 1.2.10)

“**treebank2sentences.py** extracts the sentences from a treebank. This program takes one argument : the treebank filename. It outputs the results on the standard output :

```
treebank2sentences.py Arbres > Phrases
```

2.2 Making of the models

(see section 1.3)

2.2.1 SCFG

(see section 1.3.1)

Program name : **cree-scfg.py**

This program takes two arguments : the name of the treebank file and the base name of the grammar to be written. For the experiments that indicates that there is no top level symbol of the grammar to be extracted. If this option is not passed, the program will check that there is a symbol that is never used in right-hand parts of context-free rules, and will set this symbol as the top-level symbol of the grammar.

```
cree-scfg.py -N learnArbres scfg
cree-scfg.py -N Arbres      scfgComplete
```

2.2.2 selecting trees with a maximum of 22 leaves

(see section 1.3.2)
. The command **selectArbresNbMots.py** does the job. It takes three arguments : the source treebank, the minimum number of leaves, the maximum number of leaves. It writes onto the standard output all trees of the source treebank that have a suitable number of leaves.

```
selectArbresNbMots.py learnArbres 0 22 > learnArbres22motsMax
```

2.2.3 GCFG

(see section 1.3.2)

Programs : **treebank2inout** (optionnal), **cree_GCFG**.

There are two ways of computing a GCFG. The former computes intermediate structures, called "inout" files, from the treebank, which can result in a very big files (50 GB) for the corpus. The latter computes the grammar in a single step, but can take a large amount of time. For the Wall Street Journal experiments, a parallelized version of the latter method has been designed and run on 100 machines.

with "inout" files

There are two types of "inout" files : the former, simply called "inout", represent the parse forests of the sentences of the treebank. The latter, called "arbres_inout", represent the part of the parse forests that are compatible with the actual trees of the treebank. With SCFG, the latter is only a compiled version of the treebank. But with STSGs, it represents all the decompositions of the trees in the treebank.

treebank2inout is the program that computes "inout" files from a treebank. It takes 4 parameters : the lexicon filename, the grammar filename, the "inout" base name of the files to be written (it splits the output on several 2GB files if needed), and the "arbres_inout" base name of the files to be written (same remark).

One should be be very careful to pass the same lexicon and grammar to **treebank2inout** and **cree_GCFG**.

```
treebank2inout scfgComplete.slplex scfg.slpgram \
               inout arbres\_inout \
               < learnArbres22motsMax
```

The GCFG can then be computed from those "inout" files with **cree_GCFG**, which takes 5 arguments : the lexicon filename, the grammar filename, the "inout" filename, the "arbres_inout" filename, and the base name of the grammar to be written.

The following options have been used in our experiments :

- `-gis 64` : tells the program to use the Generalized Iterative Scaling algorithm. The "64" argument indicates that there is a maximum of 64 rules in each parse. If the maximum number of rules per parse is not known, we can use "-gis 0" in order to enforce its computation.
- `-stat` : prints statistics on the standard error output.
- `-init_zero` : initially sets every potential to 0. Without this option, the program reads the probabilities of the SCFG and sets the potentials to the opposite of the logarithm of these probabilities.
- `-securite` : writes the resulting model at the end of each pass.

- `-max_passes 100` : tells the number of passes to be done.

```
cree_GCFG -gis 64 -max_passes 200 -stat -securite \
          scfgComplete.slplex scfg.slpgram \
          inout arbres_inout gcfg
```

without "inout" files

The "**cree_GCFG**" program can be used without intermediate "inout" files. For that, the third argument has to be "`-no_foret`", and the fourth argument has to be the name of the treebank file to be used for the training.

```
cree_GCFG -gis 64 -init_zero -max_passes 100 -stat \
          -securite scfgComplete.slplex scfg.slpgram \
          -no_foret learnArbres22motsMax gcfg
```

2.2.4 Head-Driven Stochastic Tree-Substitution Grammar

(see section 1.3.3)

Programs : "`extractdopgram_wsj`" and "`putdopprobamaxtrees`"

extractdopgram_wsj A Head-Driven Stochastic Tree-Substitution Grammar can be extracted from a treebank with the command "`extractdopgram_wsj`". This is a version of the standard "`extractdopgram`" of the SLPDOP toolkit. The difference lies in the fact that preterminal symbol are not hardcoded in the program. Instead of that, the correspondance between SLPToolkit codes and WSJ preterminal symbols have to be read from a file, as the one produced by the "traite_wsj.py" program.

The command takes 3 arguments : the input treebank filename, the output lexicon basename and the output grammar basename. If the lexicon already exists, the program will add new entries to the existing lexicon.

The option to produce a head-driven STSG is : `-method head <filename>`. The <filename> argument of this option is the name of the file that contains the correspondance between SLPToolkit codes and WSJ preterminal symbols. Here is a small part of such a file :

```
:13     VBG
:15     VBD
:16     VBN
:26     POS
:22     VBP
:21     WDT
:5      JJ
```

putdopprobamaxtrees The STSG extracted with "`extractdopgram_wsj`" is a proper STSG. It cannot be used as is for a Most Probable Parse polynomial search. In order to do that, we have to replace the elementary probabilities of elementary trees by their *parse*-probability. Then, the MPP search will be executed by a MPD (Most Probable Derivation) search with the modified grammar. The command for modifying the grammar is "`putdopprobamaxtrees`". It has 2 arguments : the base name of lexicon and the base name of the grammar.

Example of "extractdopgram_wsj" and "putdopprobamaxtrees" commands :

```
cree-scfg.py -N Arbres dop
extractdopgram_wsj -method head symboles.preterm \
                   learnArbres dop dop
putdopprobamaxtrees dop dop
```

(This example creates a head-driven STSG with the lexicon of the complete treebank, but with the grammar of the training treebank alone. If we do not want the complete lexicon, the first line (cree-scfg.py) has to be removed.)

2.2.5 Head-Driven Gibbsian Tree-Substitution Grammar

(see section 1.3.4)

Programs : **doptree2inout** (optionnal), **cree_GDOP**.

As for GCFGs, there are two ways of computing a GTSG. The former computes intermediate structures, called "inout" files, from the treebank, which can result in a very big files (200 GB) for the corpus. The latter computes the grammar in a single step, but can take a large amount of time. For the Wall Street Journal experiments, a parallelized version of the latter method has been designed and run on 100 machines.

with "inout" files

There are two types of "inout" files : the former, simply called "inout", represent the parse forests of the sentences of the treebank. The latter, called "arbres_inout", represent the part of the parse forests that are compatible with the actual trees of the treebank (i.e. it represents all the decompositions of the trees in the treebank).

doptree2inout is the program that computes "inout" files from a treebank. It takes 6 parameters : the lexicon filename, the CFG grammar filename, the "TSG<->CFG lexicon" filename, the filename of the lexicon of TSG rules, the "inout" base name of the files to be written (it splits the output on several 2GB files if needed), and the "arbres_inout" base name of the files to be written (same remark). It reads from its standard input the trees to be encoded.

```
doptree2inout dop.slplex \
              dop-gramCFG.slpgram \
              dop-lexCFG_DOP.slplex \
```

```
            dop-lexDOP.slplex \
            inout     arbres_inout \
    < learnArbres22motsMax
```

The GTSG can then be computed from those "inout" files with the **cree_GDOP** command, which takes 5 arguments : the lexicon base name, the grammar base name, the "inout" filename, the "arbres_inout" filename, and the base name of the grammar to be written.

The following options have been used in our experiments :

- `-gis 64` : tells the program to use the Generalized Iterative Scaling algorithm. The "64" argument indicates that there is a maximum of 64 rules in each parse. If the maximum number of rules per parse is not known, we can use "-gis 0" in order to enforce its computation.
- `-stat` : prints statistics on the standard error output.
- `-init_zero` : initially sets every potential to 0. Without this option, the program reads the probabilities of the SCFG and sets the potentials to the opposite of the logarithm of these probabilities.
- `-securite` : writes the resulting model at the end of each pass.
- `-max_passes 100` : tells the number of passes to be done.

```
cree_GDOP -gis 64 -max_passes 100 -stat -securite \
          dop dop                                 \
          inout arbres_inout                      \
          gdop
```

without "inout" files

The "**cree_GDOP**" program can be used without intermediate "inout" files. For that, the third argument has to be "`-no_foret`", and the fourth argument has to be the name of the treebank file to be used for the training.

```
cree_GDOP -securite -stat -gis 64 -securite \
          dop dop                          \
          -no_foret learnArbres22motsMax \
          gdop
```

2.2.6 Min-Max Stochastic Tree-Substitution Grammar

(see section 1.3.5)

Programs : "`extractdopgram`" and "`putdopprobamaxtrees`"

extractdopgram A Min-Max Stochastic Tree-Substitution Grammar can be extracted from a treebank with the command "`extractdopgram`" ("`extractdopgram_wsj`" can also be used for this purpose, with the same arguments).

The command takes 3 arguments : the input treebank filename, the output lexicon basename and the output grammar basename. If the lexicon already exists, the program will add new entries to the existing lexicon.

The option to produce a min-max STSG is : `-method minmax`.

putdopprobamaxtrees The STSG extracted with "`extractdopgram`" is a proper STSG. It cannot be used as is for a Most Probable Parse polynomial search. In order to do that, we have to replace the elementary probabilities of elementary trees by their *parse*-probability. Then, the MPP search will be executed by a MPD (Most Probable Derivation) search with the modified grammar. "`putdopprobamaxtrees`" is the command for modifying the grammar. It takes 2 arguments : the base name of lexicon and the base name of the grammar.

Example of "extractdopgram" and "putdopprobamaxtrees" commands :

```
extractdopgram -method minmax ../learnArbres dop dop
putdopprobamaxtrees dop dop
```

2.2.7 Min-Max Gibbsian Tree-Substitution Grammar

(see section 1.3.6)

Programs : **doptree2inout** (optionnal),
cree_GDOP or **cree_GDOP_controle_profondeur**.

The Min-Max Gibbsian Tree-Substitution Grammar can be computed exactly in the same two ways as their head-driven counterparts, provided that the grammars passed to **doptree2inout** and **cree_GDOP** are min-max STSGs : both methods (with or without "inout" files) will work.

cree_GDOP_controle_profondeur This command can replace **cree_GDOP** in order to smooth the resulting model : this version of the program uses the "Increased Depth Learning" algorithm.

The following options have been used in our experiments :

- `-gis 64` : tells the program to use the Generalized Iterative Scaling algorithm. The "64" argument indicates that there is a maximum of 64 rules in each parse. If the maximum number of rules per parse is not known, we can use "-gis 0" in order to enforce its computation.
- `-stat` : prints statistics on the standard error output.

- `-init_zero` : initially sets every potential to 0. Without this option, the program reads the probabilities of the SCFG and sets the potentials to the opposite of the logarithm of these probabilities.
- `-securite` : writes the resulting model at the end of each pass.
- `-max_passes 3` : tells the number of passes to be done *for each depth* (the real number of iterations will be this number multiplied by the maximal depth of the trees in the training treebank).

Example :

```
cree_GDOP_controle_profondeur \
                    -stat -gis 64 -securite -max_passes 3 \
                    dop dop \
                    -no_foret learnArbres22motsMax gdop
```

2.3 Parsing

(see section 1.3.7)

A few commands are useful for parsing and evaluating parsing results.

2.3.1 Parsing and evaluation with SCFGs

Standard method

The "**anagram**" program (part of SlpToolkit) is used to parse sentences with SCFGs. The experiments were conducted with the following arguments :

```
anagram -best p -input direct    \
        scfg.slplex scfg.slpgram \
        < ../testPhrases         \
        > testAnalyses
```

The comparison of the parses with the reference trees is achieved by "**compare-analyses2.py**". This program takes two arguments : the name of the reference treebank file and the name of the file *produced by anagram with the options shown in the example*. The output of "**compare-analyses2.py**" looks like this :

```
Nombre de phrases analysees                : 2416
Nombre d'analyses identiques               : 268 (11.0927152318%)
Nombre d'analyses differentes              : 2148 (88.9072847682%)
Nombre de parenthesages differents         : 2067 (85.5546357616%)
Details (nb1 - nb2, nb_cas)
-5,     3
```

```
-3,      20
-2,      75
-1,      264
0,       460
1,       588
2,       335
3,       189
4,       83
5,       32
6,       10
7,       5
9,       1
10,      1
14,      1

Esperance de la difference : 0.946298984035

Nombre d'analyses avec parenthesage croise : 1712 (70.8609271523%)

Precision : 0.866724574586

Recall    : 0.849338529382
```

From top to bottom, this shows :

- number of analysed sentences
- number of exact parses
- number of bad parses
- number of parses that have a different bracketting
- Details about bracketting differences (distribution of the difference between the number of brackets for each sentence in both files)
- likelyhood of the difference between the number of brackets in a parse of file 1 and file 2
- Number of cross bracketting parses (i.e. parses of file 2 that are not compatible with parses of file 1 in terms of bracketting)
- Label precision rate
- Label recall rate

Parallelized method

The "**anagram_pvm**" program is used to parse sentences with SCFGs in a parallelized way. It has been used for distributing the parse task amongst 100 machines. The experiments were conducted with the following arguments :

```
anagram_pvm scfg scfg            \
            < ../testPhrases     \
            >! testAnalyses
```

Notice that the `-best p` option is not required : this parallelized version of **anagram** always uses it.

The comparison of the parses with the reference trees is achieved by "**compare-analyses.py**". This program takes two arguments : the name of the reference treebank file and the name of the file *produced by* **anagram_pvm**. The output of "**compare-analyses.py**" is the same as the output of "**compare-analyses2.py**". The difference lies in the input : the output of **anagram_pvm** is : one parse per line, followed by its parenthesised score.

Warning :
Due to a bug, the first character ("_") of the output of **anagram_pvm** has to be manually removed before its use with "**compare-analyses.py**".

2.3.2 Parsing and evaluation with STSGs

Standard method

The "**analyseDOP**" program (part of slpdop) is used to parse sentences with STSGs. The experiments were conducted with the following arguments :

```
analyseDOP -MPD -input direct \
           dop dop            \
           < ../testPhrases   \
           >! testAnalyses
```

"**compare-analyses.py**" is then used as before to compare the parses with the reference trees

Parallelized method

The "**analyseDOP_pvm**" program is used to parse sentences with STSGs in a parallelized way. It has been used for distributing the parse task amongst 100 machines. The experiments were conducted with the following arguments :

```
analyseDOP_pvm dop dop          \
               < ../testPhrases
               >! testAnalyses
```

Notice that the `-MPD` option is not required : this parallelized version of **analyseDOP** always uses it.

"**compare-analyses.py**" is then used as before to compare the parses with the reference trees

Warning :

Due to a bug, the first character ("_") of the output of **anagram_pvm** has to be manually removed before its use with "**compare-analyses.py**".

Appendix A

Programs

The programs listed below allow different operations on grammars and treebanks :

- modification/adaptation/evaluation of treebanks
- visualization/selection of syntactic trees
- creation of context-free grammars
- training of gibbsian parameters for context-free and tree substitution grammars
- listing of compiled tree substitution grammars
- parallel (networked) parsing with CFGs and TSG

These programs are grouped in three directories :

- "arbres" contains C++ programs :
  - the header file "arbres.h", useful in visualization programs and also used in gibbsian training programs (i.e. all C++ programs)
  - the visualization/selection of syntactic trees
- "grammaire" contains Python programs for the textual modification of treebanks, the creation of context-free grammars, and the evaluation of parsing results
- "gdop" contains C++ programs for :
  - training gibbsian parameters for context-free and tree substitution grammars
  - listing compiled tree substitution grammars
  - parallelizing the computation of gibbsian parameters and the parsing with context-free and tree substitution grammars

A.1 Dealing with treebanks

Notes :

- All programs use procedures from file "grammaire.py".
- Except for *"traite_wsj.py"*, all programs deal with treebanks in SlpToolkit format.

A.1.1 Modifying treebanks

Directory "grammaire"

- **delexicalise_corpus.py** : replaces each terminal symbol in the treebank with the symbol that dominates it. This program takes two parameters : the source file and the destination file. Both files are in SLPToolkit format.
- **enlever_threads.py** : removes the threads from a treebank (i.e. removes rules `X->Y` by substituting `Y` to `X`). Does not remove terminal threads (attaching leaves), nor top thread (i.e. : the root of trees remains unchanged).
- **traite_wsj.py** : prepares the Wall Street Journal treebank for SlpToolkit. Delexicalization and thread removal can be done within this program by passing options. Other options available (see the self-contained help with option *-h*).
- **treebank2sentences.py** : extracts the leaves from a treebank in order to create a text file containing the associated sentences.

A.1.2 Selecting trees from a treebank

Directory "grammaire"

- **selectArbresNbMots.py** : selects the trees whose number of leaves matches a given interval. It takes three arguments : the source treebank, the minimum number of leaves, the maximum number of leaves. It writes onto the standard output all trees of the source treebank that have a suitable number of leaves.
- **separe_learn_test.py** : randomly splits one treebank into two treebanks. The ratio between the two resulting files can be chosen.
- **separe_learn_test_par_frequence.py** : splits one treebank into two treebanks. All rules appearing in the first resulting treebank will appear with a frequency greater or equal to *fmin* (see the self-contained help with option *-h*).
- **script/separation_par_nb_analyses.sh** is a little shell script that can be used for making the work with *selectarbre* easier. It will use the output of *anagram* and write several sentence files : the sentences of each file will have the same number of possible parses. The use of this script is explained in its first few lines.

Directory "arbres"

- **selectarbre** : a graphical tool for selecting trees from SlpToolkit parses. A list of sentences first have to be parsed with "*anagram*", with option "*-i p*" : the output is a file containing a list of possible parses for each sentence. **selectarbres** takes this file and lets the user choose which parse is correct for each sentence. Option "*-h*" gives the command line syntax.

A.1.3 Checking a treebank

Directory "grammaire"

- **verifie_treebank.py** : checks if a treebank is syntactically correct, i.e. in a good SlpToolkit format, readable by the other programs.

A.1.4 Evaluating parse results

Directory "grammaire"

- **compare-analyses.py** compares two treebanks and gives statistics about their differences.
- **compare-analyses2.py** also compares two treebanks, but the second one is in the format given by of *anagram -best p*.

A.1.5 Visualizing treebanks

Directory "arbres"

- **voir_arbres** : a graphical tool for the visualization of treebanks. It can take one or several treebank files as arguments. When several treebanks are given, it displays them in a parallel way, and highlights the differences by using different colors where the trees differ.

A.2 Creating context-free grammars

Directory "grammaire"

- **cree-scfg.py** : a tool that creates a stochastic context-free grammar. Takes the name of the treebank file the grammar should be extracted from, and the basename of the SCFG to be created. It writes a grammar in a textual form, then calls the SlpToolkit *compilgram* and *creelex* programs.

The following programs were heuristics to create "thermodynamic" (or gibbsian) CGFs from a standard SCFG. These methods have been replaced by the Iterative Scaling Algorithm, implemented in *cree_GCFG* and *cree_GDOP programs*.

- *cree-energie.py*
- *cree-grammaire_energie_par_queue_nulle.py*
- *cree-grammaire_proba_par_queue_1.py*
- *cree-grammaire_proba_par_queue_2.py*
- *cree-grammaire_proba_par_queue_3.py*
- *cree-grammaire_proba_par_queue_4.py*
- *cree-grammaire_proba_par_queue_et_energie_tete_nulle.py*
- *cree_grammaire_energie_moyenne_par_tete_egale_energie_tete.py*
- *cree_grammaire_energie_nulle_sur_corpus.py*
- *cree_grammaire_proba_globale.py*
- *energise_grammaire.py*
- *energise_lexique.py*

A.3 Listing a Tree Substitution Grammar

Directory "gdop/inout"

- **listgramdop** : outputs a textual representation of the elementary trees of a compiled STSG. The representation is in SlpToolkit format. The branching nodes of an elementary tree are marked with the "*" symbol.
  Example :

```
listgramdop dop.slplex dop-gramCFG.slpgram \
            dop-lexCFG_DOP.slplex dop-lexDOP.slplex
```

  An extract of the result (format : rule index - probability - rule) :

```
86037 - 4.47126e-06 - [ VP [ :13 demanding ] ]
86038 - 4.14049e-06 - [ VP [ :7 hope ] ]
86039 - 4.88742e-06 - [ VP [ VP [ :22 seem ] [ S * ] ]
[ :14 * ] [ VP * ] ]
86040 - 1.61719e-06 - [ NP [ :1 * ] [ :33 Europeans ] ]
```

A.4 Training gibbsian parameters of context-free and tree-substitution grammars

Directory "gdop/inout"

- **treebank2inout**is a program that computes "inout" files from a treebank. These "inout" files can then be used by the non-parallel version of *cree_GCFG* in order to create a gibbsian CFG. This program takes 4 parameters : the lexicon filename, the grammar filename, the "inout" base name of the files to be written (it splits the output on several 2GB files if needed), and the "arbres_inout" base name of the files to be written (same remark). It reads from its standard input the trees to be encoded. Option *-h* gives the details of available options.

- **doptree2inout** is the program that computes "inout" files from a treebank. These "inout" files can then be used by the non-parallel versions of *cree_GDOP* and its modified version *cree_GDOP_controle_profondeur* in order to create a gibbsian TSG. This program takes 6 parameters : the lexicon filename, the CFG grammar filename, the "TSG<->CFG lexicon" filename, the filename of the lexicon of TSG rules, the "inout" base name of the files to be written (it splits the output on several 2GB files if needed), and the "arbres_inout" base name of the files to be written (same remark). It reads from its standard input the trees to be encoded. Option *-h* gives the details of available options.

- **list_inout_table** is a program that writes a textual representation of CFG "inout" tables on its standard output. The form of the representation is similar to the one given by "*anagram*" without option. The listed "inout" table has to be created from a CFG, by *treebank2inout*. Command line arguments are given by the *-h* option.

- **list_dop_inout_table** is a program that writes a textual representation of TSG "inout" tables on its standard output. The form of the representation is similar to the one given by "*anagram*" without option, except that the representation of each elementary rule is in the form given by *listgramdop*. The listed "inout" table has to be created from a TSG, by *doptree2inout*. Command line arguments are given by the *-h* option.

- **compte_exemples_inout** : counts the number of "inout" structures in a file. If the file has been correctly created, this number should be the same as the number of trees in the treebank used for its creation.

- **cree_GCFG** : tool that runs IIS (Improved Iterative Scaling algorithm) or GIS (Generalized Iterative Scaling algorithm) on a SCFG. The result is a Gibbsian CFG.

  There are two ways of using this tool : if "inout" and "arbres_inout" files have been produced for a given treebank, they can be passed as arguments to *cree_GCFG*. In the other case, the program can take directly the treebank filename as argument and

dynamically produces “inout” structures when needed : with the multipass algorithms implemented by the program, this method will take much more time than the former, as “inout” structures will be recomputed at each pass.

For the former method, **cree_GCFG** takes 5 arguments : the lexicon filename, the grammar filename, the “inout” filename, the “arbres_inout” filename, and the base name of the grammar to be written.

For the latter method, **cree_GCFG** also takes 5 arguments : the lexicon filename, the grammar filename, the “-no_foret” keyword, the treebank filename, and the base name of the grammar to be written.

If *cree_GCFG* was compiled with the parallel behaviour enabled, the *-no_foret* method is mandatory, as the big “inout” structures would kill the network.

The following options are implemented :

- `-conserve_lexique` : leaves the lexicon unchanged (never updates its parameters);
- `-conserve_grammaire` : leaves the grammar unchanged (never updates its parameters);
- `-gis <n>` : tells the program to use the Generalized Iterative Scaling algorithm. The `<n>` argument indicates that there is a maximum of `<n>` rules in each parse. If the maximum number of rules per parse is not known, we can use “-gis 0” in order to enforce its computation. The GIS algorithm runs faster than the IIS algorithm because it does not make use of polynomes. However, it needs more passes to achieve the same quality of model : if “inout” structures have to be computed at each pass, this can cause the GIS method to run slower than IIS (with “inout” precomputed files, GIS uses 3 times the number of passes of IIS for the same result; but each pass is 9 times quicker.).
- `-stat` : prints statistics on the standard error output.
- `-init_zero` : initially sets every potential to 0. Without this option, the program reads the probabilities of the SCFG and sets the potentials to the opposite of the logarithm of these probabilities.
- `-init_deux` : initially sets every potential to log(2).
- `-juste_init` : only performs the initialization. To be used with `-init_zero` or `-init_deux`.
- `-securite` : writes the resulting model at the end of each pass.
- `-max_passes <n>` : tells the number of passes to be done.
- `-attend_fermeture` : if the programs is compiled with the graphical UI enabled, this option makes the monitoring window stay opened after the computation is over. Else, the monitoring window is closed as soon as the program terminates.

- **cree_GDOP** : same as *cree_GCFG* but for Tree Substitution Grammars. The arguments and options also are the same, except that :
  - the lexicon and grammar filenames have to be replaced by their *basename*, i.e. their name without any extension (.slplex / .slpgram).
  - the "inout" and "arbres_inout" files have to be produced by *doptree2inout* instead of *treebank2inout*.

  An additionnal option is also available :
  - `-prior <`σ`>` : this option will force the potentials (i.e. the gibbsian parameters of the model) to follow a gaussian prior distribution, centered on 0, with standard deviation σ. This was the first smoothing method implemented for GTSG models, but *cree_GDOP_controle_profondeur* gave much better results. However, this method could be further investigated by trying different values of σ.

  Notice that for suitable (head-driven or min-max) TSGs *putdopprobamaxtree* is intended to be used *after cree_GDOP* in order to find most probable parses in polynomial time. The most probable derivations of the resulting grammar will then be the most probable parses of the initial grammar.

- **cree_GDOP_controle_profondeur** : same as *cree_GDOP*, but implements the Increasing Depth Learning algorithm. Two more options are available :
  - `-pmax` p : fixed value to be assigned to the parameters associated to "always disambiguating" elementary trees, i.e. elementary trees that always lead to correct parses (for instance, the complete trees of the learning treebanks, in a min-max TSG). The default value of this potential is 10.
  - `-pamb` p : fixed value to be assigned to the parameters associated to "never disambiguating" elementary trees, i.e. elementary trees that appear as often in correct and incorrect parses. 0 is the default value.

  Notice that *cree_GDOP_controle_profondeur* has proved to be useful for Min-Max TSGs. For Head-Driven TSGs, the best results have been obtained with *cree_GDOP*.

A.5 parallel (networked) parsing with CFGs and TSG

Directory "gdop/inout"

The following programs are parallel versions of existing ones.

- **anagram_pvm** : performs parsing with CFGs. The arguments are the lexicon and the grammar filenames. The sentences to be parsed are read from the standard input, and the parses are written on the standard output.

  Available options :

  - `-P` : Only looks for parses starting with the top level symbol.

  The other options of *anagram* have not been implemented. The behaviour is the same as *anagram -best p -input direct*, except for the format of the output : it only outputs one parse per line, followed by its probability.

- **analyseDOP_pvm** : performs parsing with TSGs. The arguments are the lexicon and the grammar filenames. The sentences to be parsed are read from the standard input, and the parses are written on the standard output.

  Available options :

  - `-P` : Only looks for parses starting with the top level symbol.
  - `-normlex` : normalizes the lexicon probabilities (for a standard STSG, not to be used with GTSGs)
  - `-normgram` : normalizes the grammar probabilities (for a standard STSG, not to be used with GTSGs)

  The other options of *analyseDOP* have not been implemented. The behaviour is the same as *analyseDOP -MPD -input direct*, except for the format of the output : it only outputs one parse per line, followed by its probability.

A.6 Implementation notes

This section only concernes the C++ programs of directory *gdop/inout*.

A.6.1 Five parallel programs

"**anagram_pvm**" and "**analyseDOP_pvm**" are parallelized versions of "anagram" and "analyseDOP".

Concerning "cree_GCFG", "cree_GDOP" and "cree_GDOP_controle_profondeur", the way they work depends on the options used at compilation time. In order to run in a parallel way, they have to be compiled with the option "`pvm=oui`" of the "make" command.

The parallel versions work with `PVM 3.4`. PVM stands for Parallel Virtual Machine. This program has to be installed on each machine that will participate in the computation task.

Moreover, two other programs have to be placed in the "bin" directory of the PVM architecture on each machine : these programs ("**clientInout**" and "**surveillant**") contain the *client* part required for the five programs to run in a parallel fashion.

When everything is properly installed, the PVM has to be started before running one of the five programs.

Example :

```
pvm hostsIn1In3
^D
```

```
anagram_pvm scfg scfg            \
            < ../testPhrases     \
            >! testAnalyses
```

The first line of the example starts a PVM spread on all machines listed in the file "`hostsIn1In3`".

The second line (`ctrl-D`) is used for exiting the PVM console without shutting down the PVM.

The last commamd will launch *anagram_pvm*. This program will connect to the PVM and will launch the clients (*clientInout* and *surveillant*) on each host of the PVM, before starting the actual parsing task.

Warning : The programs currently filter the hosts : they launch computations only on hosts that begin with a name lexicographically smaller than "lia". This artificial filtering has been done in order to launch the main program from a LIA's machine, but to restrict the heavy computation to In1 and In3 machines.

This filtering is done in the "*init_clients()*" method of "*inout_gibbs.cc*" file.

A.6.2 Monitoring PVM programs

- **moniteur_pvm** is a tool for monitoring PVM programs. It tries to connect to the PVM and sends requests for information. The information it receives is then displayed on a graphical window. This tool is useful for looking at the progress of PVM programs. All five mentionned programs will reply to *moniteur_pvm* and will send the requested information.

A.6.3 Compiling the C++ programs

The Makefile can take three options :

option	possible values
level	debug optimize O3
qt	oui non (default)
pvm	oui non (default)

The *level* option is self-explanatory for gcc users.

If the "*qt=oui*" option is passed to the *make* command, some programs will implement a graphical monitoring window :

- *cree_GCFG*,
- *cree_GDOP*,
- and *cree_GDOP_controle_profondeur*.

This option must be avoided for the compilation of *clientInout* and *surveillant* : as these programs are intended to run on distant machines, they will likely be unable to open a local graphical window.

The "*pvm=oui*" option is required for compiling the five PVM programs, and also for compiling *clientInout*, *surveillant* and *moniteur_pvm*. When this option is used, the paths to PVM 3.4 and PVM++ libraries must be correct in the Makefile.

The versions of the libraries used for compiling the programs are :

- qt-x11-free-3.1.2
- PVM 3.4 (I have a modified version of these libraries)
- PVM++ 0.6.0

A.6.4 SlpToolkit and the Wall Street Journal corpus

Most C++ programs dynamically depend on the SlpToolkit library. For the first series of experiments (i.e. with a lexicalized version of the WSJ treebank), the sizes of used SlpToolkit lexicons are very big. This means that the SlpToolkit library **have to be compiled with the macro** `LONG_DEPL_SIZE` **defined**. This can be done in the `arbregen.h` file, before the `DEPL_SIZE` declarations.