

# 3.3: Implementation of Tree Transfer System

Ondřej Bojar, Miroslav Janíček, Miroslav Týnovský

Distribution: Public

**EuroMatrix** Statistical and Hybrid Machine Translation Between All European Languages IST 034291 Deliverable 3.3

September, 2008



Project funded by the European Community under the Sixth Framework Programme for Research and Technological Development.



Project ref no.	IST-034291
Project acronym	EuroMatrix
Project full title	Statistical and Hybrid Machine Translation Between All Eu-
	ropean Languages
Instrument	STREP
Thematic Priority	Information Society Technologies
Start date / duration	01 September 2006 / 30 Months

Distribution	Public
Contractual date of delivery	August, 2008
Actual date of delivery	September, 2008
Deliverable number	3.3
Deliverable title	Implementation of Tree Transfer System
Туре	
Status & version	
Number of pages	47
Contributing $WP(s)$	3
WP / Task responsible	Jan Hajič
Other contributors	
Author(s)	Ondřej Bojar, Miroslav Janíček, Miroslav Týnovský
EC project officer	Xavier Gros
Keywords	

The partners in EUROMATRIX are:	Saarland University (USAAR)
	University of Edinburgh (UEDIN)
	Charles University (CUNI-MFF)
	CELCT
	GROUP Technologies
	MorphoLogic

For copies of reports, updates on project activities and other EUROMATRIX-related information, contact: The EUROMATRIX Project Co-ordinator Prof. Hans Uszkoreit Universität des Saarlandes, Computerlinguistik Postfach 15 11 50 66041 Saarbrücken, Germany uszkoreit@coli.uni-sb.de Phone +49 (681) 302-4115- Fax +49 (681) 302-4700

Copies of reports and other material can also be accessed via the project's homepage: http://www.euromatrix.net/

### © 2007–2008, The Individual Authors

No part of this document may be reproduced or transmitted in any form, or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission from the copyright owner.

## Contents

1	$\operatorname{Intr}$	oduction	5				
<b>2</b>	Tree	ree Aligner 6					
	2.1	Users' Manual	6				
		2.1.1 Demo for Java Implementation	6				
		2.1.2 Demo for Perl Implementation	7				
		2.1.3 Input Data Format	8				
	2.2	Overview of Alignment Algorithms	9				
		2.2.1 Safe Order of Synchronous Rules	9				
		2.2.2 Serialization of Rules	0				
		2.2.3 EM Iteration $\ldots \ldots \ldots$	0				
		2.2.4 Backoff Models	1				
	2.3	Detailed Configuration Options	2				
		2.3.1 Configuring Java Implementation	2				
		2.3.2 Configuring Perl Implementation	3				
	2.4	Technical Issues	3				
		2.4.1 Detaching Lexical Information	3				
		2.4.2 Estimation of the Number of Extractable Treelets	4				
		2.4.3 Branching Factor Estimation	5				
		2.4.4 Pruning of Rules with a Low Probability	6				
		2.4.5 Pruning of Low Frequency Treelets	6				
		2.4.6 Pruning Using Word-Alignment	6				
		2.4.7 Experiments with Pruning Methods	7				
	2.5	Remarks on Perl Reimplementation	8				
		2.5.1 Motivation	8				
		2.5.2 Modules Documentation	8				
	2.6	Future Plans	9				
	2.0	2.6.1 Using External Storage	9				
		2.6.2 Improving Pruning Methods and Combining with Backoff Models	9				
		2.6.3 Impact on TreeDecode	9				
3	Tree	e Decoder: TreeDecode 20	)				
	3.1	General Overview and Features	0				
		3.1.1 TREEDECODE Operation	U				
		3.1.2 Parallelization and Caching	1				
	3.2	Users' Manual $\ldots \ldots 2$	2				
		$3.2.1  \text{Installation}  \dots  \dots  \dots  \dots  \dots  \dots  \dots  \dots  \dots  $	2				
		3.2.2 Quick Run	2				
		3.2.3 Configuring an Experiment	3				
		3.2.4 Running TREEDECODE	1				
		3.2.5 Troubleshooting $\ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots 3$	4				
	3.3	Input Data Formats	4				
	3.4	Experimental Loop	5				

		3.4.1	Starting an Experiment	35
		3.4.2	Working Directory of an Experiment	36
		3.4.3	Cloning an Experiment	36
		3.4.4	Giving Experiments Symbolic Names	36
		3.4.5	Collecting Results from Many Experiments	36
	3.5	Source	Code Overview	36
		3.5.1	Libraries Used	37
		3.5.2	Description of Core TREEDECODE Modules	37
		3.5.3	Wishlist of Future Improvements	40
Α	Refe	erences		42

### B Example TreeDecode Configuration File

**43** 

# Chapter 1 Introduction

This document describes our implementation of the tree-transfer system as proposed in Bojar and Čmejrek (2007). The reader is assumed to be familiar with the theoretical background.

Here we cover both the users' manual as well as a brief overview of the source code for software developers interested in re-using our code. The users' manual includes the installation and configuration process, command-line options and the structure of input and configuration files. We also mention some basic experiments carried out with our system.

Chapter 2 is devoted to the tree alignment system, a tool for expectation-maximization (EM) training of Synchronous Tree Substitution Grammars (STSG), including a discussion of current problems with memory requirements of the model. Chapter 3 covers tree decoder, a system to translate source dependency trees into dependency trees or strings in the target language.

### Chapter 2

## Tree Aligner

### 2.1 Users' Manual

We describe two implementations of a tree aligner based on Synchronous Tree Substitution Grammar (STSG). The first one is the original Java implementation by Čmejrek (2006). The only changes we made to this implementation are (1) loading a different input file format and (2) using additional pruning methods. The second implementation is a complete rewrite in Perl which we made in order to decrease memory requirements and to abridge the source code. Both of the implementations were developed and tested under Linux, however they should be easily portable to other platforms.

Sections 2.1.1 and 2.1.2 contain a quick introduction to both of the implementations and step-by-step instructions to get working demos. Section 2.2 provides an overview of the alignment algorithms for two reasons: (1) to bridge the gap between the theoretical description in Bojar and Čmejrek (2007) and the implementation, and (2) to lay foundations for the full configuration possibilities of the two implementations in Sections 2.3.1 and 2.3.2.

### 2.1.1 Demo for Java Implementation

The demo of the Java implementation generates an STSG grammar based on 10 pairs of input sentences and prints their most likely ("Viterbi") alignments. To get a working demo of the Java implementation, follow these few steps.

- 1. Install dependencies: you will need to install Sun Java Development Kit, and Apache Ant building tool.
- 2. Get the package treeali-0.8-java.tgz containing the source codes and the demo input files from the following page:

http://ufal.mff.cuni.cz/euromatrix/

3. To compile and build the system, run **ant** in the system root directory:

```
tar xzf treeali-0.8-java.tgz
cd treeali-0.8-java
ant
```

4. Now, everything is prepared to run the demo. Go to the demo directory and check out its content:

cd demo ls There are several files:

- demo.cs, demo.en—these are the input files containing tree pairs
- demo.properties—a demo configuration file
- iteration0.sh, iteration\_loop.sh—executable scripts
- 5. Run the initialization—observe synchronous rules in the input tree pairs

./iteration0.sh

The files demo0.log and model0 are created.

6. Run the Expectation Maximization loop (10 iterations)

./iteration\_loop.sh

Rule probabilities are stored in the binary files model[0-10] (one for each iteration) and the most likely alignments for all tree pairs are printed to standard output.

Each of the most likely alignments consists of treelet pairs which cover the whole input tree pair. These treelet pairs are listed in a linearized form. Each line stands for a single treelet pair. Left (Czech) and right (English) treelets are separated by four dashes ----. The treelets are represented as lists of nodes. For each node, we print the node type (I for internal, F for frontier), the index in the original sentence and the lexical value. An example of the most likely alignment of one tree pair follows:

```
sentence1: dnes myslit_si ten být sen
sentence2: #PersPron seem dream #PersPron today
I,2,myslit_si I,4,být F,1,dnes F,3,ten F,5,sen
---- I,2,seem I,3,dream F,1,#PersPron F,4,#PersPron F,5,today
I,5,sen ---- I,4,#PersPron
I,3,ten ---- I,1,#PersPron
I,1,dnes ---- I,5,today
```

The tree structure is not included in this simple notation but the treelets are correct subtrees of the input tree. To reconstruct the structure of the treelets, consult the input corpus files (Section 2.1.3).

### 2.1.2 Demo for Perl Implementation

The demo of the Perl implementation does the same as the Java demo, i.e. it generates an STSG grammar based on 10 pairs of input sentences and prints their most likely alignments. To get a working demo of the Perl implementation, follow these few steps.

- 1. Install dependencies: you will need Perl (the implementation was tested on Perl version 5.8.8) and Perl modules Config::Tiny, Data::Dumper, BSD::Resource and Storable.
- 2. Get the package treeali-0.8-perl.tgz containing the source codes and the demo input files from the following page:

http://ufal.mff.cuni.cz/euromatrix/

- 3. Check out the content of the directory treeali-0.8-perl, there are several files there:
  - demo.cs, demo.en—input files containing tree pairs, identical to the input files in the Java demo,

- pstsg.conf—a demo configuration file.
- \*.pm—Perl modules implementing STSG,
- observe\_synchronous\_rules.pl—implementation of initial extraction of synchronous rules from tree pairs,
- em-iteration.pl—Perl implementation of an iteration of Expectation. Maximization loop for refining of rule probabilities,
- em-loop.sh—a script for executing em-iteration.pl in a loop,
- iteration0.sh—a script for executing observe\_synchronous\_rules.pl on 10 tree pairs of input files,
- 4. Everything is prepared, we can run the initial step:

```
tar xzf treeali-0.8-perl.tgz
cd treeali-0.8-perl
./iteration0.sh
```

This step will create the file model0 containing observed synchronous rules with initial probabilities.

5. ... and the Expectation Maximization loop:

./em\_loop.sh

This creates the files model [0-10], one for each iteration. Additionally, the most likely treelet alignments are printed to the standard output in a linearized form as described in Section 2.2.2 below.

### 2.1.3 Input Data Format

Input tree pairs are stored in a simple line-based format, each line corresponds to a tree. There is one file for each language, the sentence-level alignment is given implicitly by the order of lines. On every line, the tokens are separated by single spaces. The tokens are six-tuples of "factors", separated by the vertical bar symbol (1).

The use of the first three factors depends on the language, level of annotation and specifics of the model, the 4<sup>th</sup> to the 6<sup>th</sup> factor represent the tree structure and the word alignment: the linear order of the node in the sentence, the index of the parent node and the index of the corresponding node in the other tree (we assume at most 1-1 node alignment coming e.g. from an intersection of two 1-n alignments).

Figure 2.1 provides a sample pair of sentences, their representation on the tectogrammatical layer (Mikulová et al., 2007) and the factored notation to express the tectogrammatical trees in the source files. See Table 2.1 for the interpretation of factors used in our example.

Factor Index	Czech	English	
1	base form or t-lemma of the word		
2	semantic part of speech	functor	
3	formeme		
4	word alignment		
5	parent token		
6	word ord	ler	

Table 2.1: A possible usage of factors in Czech and English tectogrammatical trees.

	Czech	English	
Original Sentence	Jan má rád Marii.	John likes Mary	
Tectogrammatical tree	t-lemma: mít sempos: v formem: v:fin t-lemma: jan t-lemma: rád t-lemma: mary sempos: n.denot sempos: adj.denot sempos: n.denot formem: n:1 formem: adj:compl formem: n:4	t-lemma: like functor: PRED formem: v:fin t-lemma: john t-lemma: mary functor: ACT functor: PAT formem: n:subj formem: n:obj	
Factored notation	jan n.denot n:1 1 2 1 mít v v:fin 0 0 2 rád adj.denot adj:comp1 2 2 3 marie n.denot n:4 3 2 4	john ACT n:subj 1 2 1 like PRED v:fin 3 0 2 mary PAT n:obj 4 2 3	

Figure 2.1: A sample input tree pair.

Al	Algorithm 1 Observing all synchronous treelets usable for tree parsing.			
1.	function observe synchronous rules(left_tree, right_tree)			
2.	$left\_treelets \leftarrow extract all treelets from left\_tree$			
3.	prune left_treelets			
4.	right_treelets $\leftarrow$ extract all treelets from right_tree			
5.	prune right_treelets			
6.	rule_templates $\leftarrow$ make pairs of left_treelets and right_treelets (all combinations)			
7.	prune rule_templates			
8.	rules $\leftarrow$ augment rule_templates with all possible frontier mappings (pruned)			
9.	return rules			
10.	end			

### 2.2 Overview of Alignment Algorithms

A (synchronous) rule in STSG consists of a pair of treelets and a mapping of their frontier nodes. The task of finding a treelet-to-treelet alignment is the task of finding a set of rules that synchronously generate the left and the right tree. We can also say that the left and right trees are synchronously decomposed into treelets which are the left and right sides of synchronous rules and attached at frontiers linked by the frontier mapping. The treelet-to-treelet alignment is implied by the synchronous decomposition.

The method we use to align subtrees of the original pair of trees is an application of Expectation Maximization (EM) algorithm analogical to Inside-Outside algorithm for training Probabilistic Context-Free Grammars (PCFGs). In each iteration, we count the inside and outside probabilities for every node pair of every input tree pair using the model generated by the previous iteration. Using these probabilities, we update expected counts of all rules and use them to estimate rule-probabilities of the new model. The formulas for computations of inside probabilities, outside probabilities can be found in Bojar and Čmejrek (2007).

The detailed procedure is described in Algorithms 2 and 3 that rely on a common subroutine (Alg. 1). Alg. 2 describes the "zeroth" iteration—observing all the synchronous rules potentially usable for parsing input tree pairs. Alg. 3 is one iteration of the EM algorithm. Alg. 3 is repeated until convergence. The formal description in Bojar and Čmejrek (2007) shows that the attachment of a treelet into a frontier node with different state is allowed in the derivation step. Algorithms 2 and 3 do not use this derivation possibility as we do not add any rules not seen in the data.

### 2.2.1 Safe Order of Synchronous Rules

When computing the inside probabilities in Alg. 3, the rules must be processed in a specific order so that inside probabilities of all treelets attachable to the current rule have already been

### Algorithm 2 Compution of initial rule probabilities.

1.	for each (left_tree, right_tree) $\leftarrow$ tree-pair from input file
2.	$Rs \leftarrow observe synchronous rules(left_tree, right_tree)$
3.	for each rule $R \in Rs$
4.	serialization $\leftarrow$ serialize R
5.	increment expected count of serialization
6.	end
7.	end
8.	for each rule $R \in Rs$
9.	compute probability of R out of its expected count
10.	end

Al	gorithm 3 EM re-estimation of the probabilities.
1.	for each (left_tree, right_tree) $\leftarrow$ tree-pair from input file
2.	$Rs \leftarrow observe synchronous rules(left_tree, right_tree)$
3.	for each rule $R \in Rs$ in safe order
4.	count inside probability update of R for the chart cell
5.	determined by left and right treelet bottom up indices
6.	end
7.	for each rule $R \in Rs$ in reversed safe order
8.	count outside probability update of R for the chart cell
9.	determined by left and right treelet bottom up indices
10.	end
11.	for each rule $R \in Rs$
12.	count expected count update of R
13.	end
14.	end

estimated. We call this order "safe". Every safe order of rules follows bottom-up ordering of nodes in the original trees, so we represent nodes in input trees using indices assigned from the bottom to the top. Moreover, using a node as an internal always has to precede all rules that use the node as frontier.

Table 2.2 lists all treelets generated independently from the left and right trees in Figure 2.1 in a safe order. The Cartesian product of these treelets makes the set of "rule templates". If the Cartesian product preserves the order of both input lists, the resulting rule templates are in safe order, too. Safe order of rule templates ensures safe order of the rules—the frontier pairings of each template can be ordered arbitrarily.

### 2.2.2 Serialization of Rules

The rules are stored in this serialized form: (v  $\rightarrow$  (Ijan)Imít(Fadj.denot)(Fn.denot))(PAT  $\rightarrow$  FPAT)(1-1)

- (v  $\rightarrow$  (Ijan)Imít(Fadj.denot)(Fn.denot)) represents the left treelet
- (PAT  $\rightarrow$  FPAT) represents the right treelet
- (1-1) represents the frontier mapping of Fadj.denot and FPAT, the remaining frontier node (Fn.denot) is mapped to the null tree

### 2.2.3 EM Iteration

A chart is constructed for each input tree pair. A chart is a matrix with rows numbered by left tree nodes' bottom-up indices and columns numbered by right tree nodes' bottom-up indices. A cell [i, j] contains inside and outside probabilities of the subtree pair rooted in nodes with bottom up indices i and j from left and right input trees respectively.

	Left treelets		Right treelets
1	$nullState \rightarrow nullLabel$	1	$nullState \rightarrow nullLabel$
2	$n.denot \rightarrow Ijan$	2	$ACT \rightarrow Ijohn$
3	$n.denot \rightarrow Fn.denot$	3	$ACT \rightarrow FACT$
4	$adj.denot \rightarrow Irád$	4	$PAT \rightarrow Imary$
5	$adj.denot \rightarrow Fadj.denot$	5	$PAT \rightarrow FPAT$
6	$n.denot \rightarrow Imarie$	6	$PRED \rightarrow (Ijohn)Ilike(FPAT)$
7	$n.denot \rightarrow Fn.denot$	7	$PRED \rightarrow (FACT)Ilike(Imary)$
8	$v \rightarrow (Ijan)Imít(Fadj.denot)(Fn.denot)$	8	$PRED \rightarrow (FACT)Ilike(FPAT)$
9	$v \rightarrow (Fn.denot)Imit(Irád)(Fn.denot)$	9	$PRED \rightarrow FPRED$
10	$v \rightarrow (Fn.denot)Imit(Fadj.denot)(Imarie)$		
11	$v \rightarrow (Fn.denot)Imit(Fadj.denot)(Fn.denot)$		
12	$v \rightarrow Fv$		

Table 2.2: Left and right treelets generated in a safe order from the sample tree pair in Figure 2.1. Semantic parts of speech are used as Czech frontier states, functors as English frontier states, tectogrammatical lemmas as both Czech and English internal node labels.

As can be seen in Alg. 3, all rules are iterated three times to update inside probabilities, outside probabilities and to compute expected counts. Figure 2.2 shows an example of a cell's inside probability update by a rule. The safe order of rules ensures that the chart cells used for update computation have already been finished. The update of outside probability estimates and expected counts is done analogically.



Figure 2.2: Example computation of inside probability update by a rule. The cell [4,3] corresponding to the pair of roots is updated using a grammar rule and the cells [1,1] and [3,2].

### 2.2.4 Backoff Models

A big problem of training STSG is huge data sparseness. The solution proposed by Čmejrek (2006) is to use backoff models, i.e. models which use less fine-grained rule descriptions. The granularity is defined by the following features:

- 1. the allowed size of the treelets, i.e. the number of frontier and internal nodes
- 2. the information stored in internal node labels,
- 3. the information stored in frontier states,
- 4. whether the tree structure is encoded in the rule or not.

We limit the maximum treelet size to at most 2 internal nodes and at most 5 frontier nodes. The label and state strings are constructed from the three non-structural input factors, see Section 2.1.3. If the tree structure is omitted, treelets become just sequences of nodes. Perl and Java implementations differ in possibilities of backoff models definitions. There are several predefined backoff models in the Java implementation whereas it is possible to directly choose the granularity of internal labels, frontier states and tree structure in the Perl implementation. Table 2.3 lists the predefined backoff models with the corresponding feature settings.

	Factor 1	Factor 2	Factor 3	
Predefined	(Czech t-lemma)	(Czech formem)	(Czech sempos)	Structure
backoff model	(English t-lemma)	(English formem)	(English functor)	preserved
Full Model	label	label	state	YES
Linear	label	label	state	NO
Linear Function				
and Morphology	ignored	label	state	NO
Linear Function				
and Function	ignored	ignored	label, state	NO
Structure	ignored	ignored	ignored	YES

Table 2.3: Predefined backoff models of the Java implementation. Each cell in the table specifies which factor is used where, e.g. the "Linear Function and Morphology" model ignores factor 1, uses factor 2 as the label of internal nodes and factor 3 as the state of frontier nodes.

### 2.3 Detailed Configuration Options

### 2.3.1 Configuring Java Implementation

The Java implementation of the aligner gets configuration from various sources: a configuration file, command line options given to the aligner, command line options given to Java Virtual Machine (JVM), and finally the input data used. We list the basic options and show how to set them.

Training Data: command line options --leftTrees and --rightTrees given to aligner.

- Subset of the Training Data: command line options --startFromSentence and --sentences given to aligner. These options say which sentence to start from (the sentences are numbered from 0) and how many sentences to use for the training.
- **Backoff Models:** the command line option --bm given to the aligner. The option can be repeated to specify several models. The following backoff models are supported:
  - FULL—Full model,
  - FM—Linear Function and Morfology model
  - FF—Linear Function and Function model
  - S—Structure model

**Pruning Method:** the configuration file options initialPruning and iterationPruning. The following pruning methods are supported, for a description see Section 2.4.6:

- TreeletsReflectWordAlignment
- InternalsReflectWordAlignment
- FrontierMappingsReflectWordAlignmentStrict
- FrontierMappingsReflectWordAlignmentBenevolent

Amount of Virtual Memory Space: command-line option -Xmx to the JVM.

Model Filename Prefix: command-line option --modelFN given to aligner.

For an example of the configuration file options, see the file demo.properties and for an example of the command-line options see the file demo.sh both in the Java demo directory.

### 2.3.2 Configuring Perl Implementation

The Perl implementation gets configuration options from command line and from configuration file. There are only two command-line options; --iteration specifies the number of iteration of EM loop and --config specifies the configuration file containing all other options:

Backoff Models are specified by backoff\_models option. Its value are space separated backoff model codes. Each code is in form of a-b-c, where

- a is a set of factor indices which determine frontier node state
- b is a set of factor indices which determine internal node label
- c is a boolean determining whether treelet structure is preserved

## **Pruning Methods** are specified by left\_treelet\_pruning, right\_treelet\_pruning, treelet\_pairs\_pruning and rule\_pruning options.

Amount of Virtual Memory Space is specified by virtual\_memory\_size option.

Model Filename Prefix is specified by model\_prefix option.

For an example of configuration file see the file pstsg.conf in the Perl demo directory.

### 2.4 Technical Issues

In the following, we describe several technical difficulties with the Java implementation of the aligner and we examine ways of solving the problems.

The first simple problem was the format of input tree pairs. The original implementation used a pair of  $CSTS^1$  files each containing trees for one language. We avoided the conversion of our dataset into CSTS because of character encoding issues and changed the implementation to accept the simple line-based input format as described in Section 2.1.3.

A more complex problem are immodest memory requirements of the software. We proposed several independent ways to decrease the memory requirements. All the ways can be applied together to get the best results.

In general, there are two ways of memory savings:

- decreasing the size of each rule—this method is discussed in Section 2.4.1,
- decreasing the number of stored rules—this method is discussed in Sections 2.4.2, 2.4.4, 2.4.5 and 2.4.6.

### 2.4.1 Detaching Lexical Information

We can save memory by decreasing the size of each rule stored. As the rules are stored in a serialized form, our goal is to decrease the length of the serializations. We do this by detaching lexical information into separated dictionaries. In other words, we create arrays of all possible labels and states of the input nodes and store indices to the arrays instead of full strings to the rules. As shown in Table 2.4, this shortens the length of a rule on average by 33 %. Further reduction could be achieved by indexing lexical items in the order of decreasing frequency and by using some variable-length encoding of integers, e.g. UTF-like notation or Elias gamma code.

 $<sup>^{1}</sup>$ CSTS is an obsolete format for storing dependency trees, used in the Prague Dependency Treebank 1.0 and Prague Czech-English Dependency Treebank 1.0.

Lexical information	Rules extracted	Tree pairs processed
full	$8,\!612,\!247$	298
detached	$12,\!993,\!863$	419
stripped	$16,\!848,\!309$	646

Table 2.4: Comparison of rule counts that fit into 4 GB of memory depending on the method of storing lexical information in rules. The third line shows the upper bound: the number of rules with lexical information completely removed.

### 2.4.2 Estimation of the Number of Extractable Treelets

Given a tree pair of m and n nodes on the left and right sides respectively, let us estimate an upper bound on the number of rules that can be extracted.

The number of treelets extracted from the left tree is sum of these counts:

- 1 null treelet,
- *m* treelets containing a single frontier node (and no internal node),
- *m* treelets containing a single internal node with all its children as frontier nodes,
- m-1 treelets containing two internal nodes and all their children as frontier nodes (we do not allow more than two internal nodes in the treelets).

To get the number of rules, we need to count all frontier nodes mapping of each treelet pair. If  $f_l$  and  $f_r$  are the frontier node counts in left and right treelets respectively, the number of mappings p can be computed using this formula:

$$p(f_l, f_r) = \sum_{k=0}^{\min(f_l, f_r)} {\binom{f_l}{k}} \frac{f_r!}{(f_r - k)!}$$
(2.1)

- k is the number of edges in the mapping, ranging from 0 to  $min(f_l, f_r)$  when all frontiers in one of the treelets are mapped,
- $\binom{f_l}{k}$  is the number of all possible subsets of left frontier nodes that will be mapped to a right frontier node (the remaining left frontiers are left unmapped),
- $\frac{f_r!}{(f_r-k)!}$  is the number of possible assignments of right frontier nodes to the selected k left frontier nodes. (Every linked left node gets its counterpart from the right nodes that has not been used yet.)

Assuming the maximum branching factor is b, one-internal-node treelets contain at most b frontiers and two-internal-node treelets contain at most 2b - 1 frontiers. Finally, null treelets contain no frontier and single-frontier-node treelets contain one frontier.

To estimate the upper bound on extractable treelets, we assume the maximum branching for every node. The number of extractable treelets is then the sum of:

- $(m-1)(n-1) \cdot p(2b-1, 2b-1)$  for pairs of two-internal-node treelets,
- $m(n-1) \cdot p(b, 2b-1)$  for pairs (one-internal-node treelet, two-internal-node treelet),
- $(m-1)n \cdot p(2b-1,b)$  for pairs (two-internal-node treelet, one-internal-node treelet),
- $mn \cdot p(b, b)$  for pairs of one-internal-node treelets,
- $m(n-1) \cdot p(1, 2b-1)$  for pairs (one-frontier-node treelet, two-internal-node treelet),
- $(m-1)n \cdot p(2b-1,1)$  for pairs (two-internal-node treelet, one-frontier-node treelet),

- $mn \cdot p(1, b)$  for pairs (one-frontier-node treelet, one-internal-node treelet),
- $mn \cdot p(b, 1)$  for pairs (one-internal-node treelet, one-frontier-node treelet),
- $(n-1) \cdot p(0, 2b-1)$  for pairs (null treelet, two-internal-node treelet),
- $(m-1) \cdot p(2b-1,0)$  for pairs (two-internal-node treelet, null treelet),
- $n \cdot p(0, b)$  for pairs (null treelet, one-internal-node treelet),
- $m \cdot p(b,0)$  for pairs (one-internal-node treelet, null treelet),
- $n \cdot p(0,0)$  for pairs (null treelet, one-frontier-node treelet),
- $m \cdot p(0,0)$  for pairs (one-frontier-node treelet, null treelet),
- $m \cdot p(0,0)$  for pairs of one-frontier-node treelets,
- p(0,0) for a pair of null treelets.

In our experiments, we restrict the branching factor to value 5. All trees containing a node with 6 or more children are not processed. With this branching factor and assuming 20 words per sentence, our formula gives us about  $63 \cdot 10^9$  rules. This worst case estimate is obviously very pessimistic as not all nodes have the maximum branching factor, but if we assign b=3, the estimate still remains unfavourable giving 601,585 rules extractable from a single tree pair.

### 2.4.3 Branching Factor Estimation

We can estimate the average branching more precisely by exploration of our training data: the parallel corpus CzEng (Bojar et al., 2008) containing 1 million sentences in both Czech and English. The distributions of nodes' branching factors in Czech and English parts of input data are shown in Table 2.5 and Table 2.6 respectively.

Branching factor	0	1	2	3	4	5	6 - 50
Thousands of nodes	4,935	3,030	1,572	629	213	56	21

Table 2.5: Branchin	g factor	distribution	in	Czech	data.
---------------------	----------	--------------	----	-------	-------

Branching factor	0	1	2	3	4	5	6 - 50	51 - 3561
Thousands of nodes	6,198	2,286	1,523	681	265	91	64	7

Table 2.6:	Branching	factor	distribution	$\mathrm{in}$	English	data.
------------	-----------	--------	--------------	---------------	---------	-------

To get an upper and lower estimate of total count of rules extractable from the input data, we find maximal branching factor in each tree. The lower estimate is then calculated as number of rules generated by treelet pair composed of a single-internal-node treelet on each side, where the internal node has the maximal branching factor (i.e. a subset of all rules generated by the tree pair). The upper estimate is calculated assuming each node in a tree has the maximal branching factor of the tree and summing similarly as in previous section.

Based on CzEng data, the upper estimate of rule count per sentence is  $2, 1 \cdot 10^9$  and the lower estimate per sentence is 1560. If we assumed absolutely no lexical overlap between various sentences, the extracted grammar would need to contain something between  $1.5 \cdot 10^9$  and  $2.1 \cdot 10^{15}$  rules. Though quite imprecise, these estimations show that the number of extractable rules is huge and a pruning method is necessary.

### 2.4.4 Pruning of Rules with a Low Probability

A very straightforward pruning method is to disqualify rules which obtain a very low probability after a few iterations of the EM algorithm. This can be achieved by setting a threshold of some minimum required probability. Another possibility is to keep only a fixed number of the most probable rules for each left hand side of rule (root state).

This approach still has several problems. First, the pruning is applicable only after a few iterations so memory requirements of the first few iterations stay unreduced. Second, the pruning must be in general less restrictive than in e.g. phrase-based machine translation because the set of rules has to cover the whole training corpus whereas in phrase-based machine translation, there is only one sentence affected by the pruning at a moment. Additionally, a single threshold is hardly applicable for conditional probabilities: too many rules would be pruned for frequent root states, too few rules would be pruned for rare root states.

### 2.4.5 Pruning of Low Frequency Treelets

Pruning of low frequency treelets is a similar method but it does not suffer from the listed problems. This method requires a prior processing of both sides of the training parallel corpus. All one-side treelets are extracted and counts of their occurrences are stored. In the phase of observing synchronous rules, only rules consisting of frequent treelets are preserved.

Note that in contrast to previous pruning method, this one gives benefit of memory savings as the rules are pruned even in the "zeroth iteration". Additionally, the pruning is well motivated: the rules with low frequency treelets could be marginal or erroneous.

### 2.4.6 Pruning Using Word-Alignment

Another pruning methods take into account information of word alignment for disqualifying rules. We use the intersection of GIZA++ Czech-English and English-Czech word alignments. We distinguish several pruning methods:

- TreeletsReflectWali is the most straightforward pruning method. It disqualifies each rule which contains a node (on left or right side) aligned to a node outside of the rule (i.e. all word-alignment edges have to stay within the left and right treelets).
- InternalsReflectWali is similar to the previous method, except that only alignments starting in internal nodes are considered. A word-alignment link starting in a frontier node and leading outside of the rule thus does not prevent the extraction.
- FrontierMappingsReflectWaliStrict restricts allowed rules on the basis of frontier mapping.
  If the mapping specifies a link between two frontier nodes, the nodes have to be aligned
  using the word alignment, too.
- FrontierMappingsReflectWaliBenevolent is a more benevolent variant of the previous method. It allows a frontier node to be word-aligned to any node in the subtree of the paired frontier node.

Unfortunately, the combination of the limit of treelet size and word-alignment pruning can completely prohibit the synchronous parsing. Figure 2.3 shows schemas of two situations which lead to such inappropriate pruning.

Consider Figure 2.3 (a) and the pruning method InternalsReflectWali. For contradiction, suppose there exists a synchronous decomposition respecting the indicated word-alignment and containing at most 2 internal nodes in every treelet. Due to the treelet size limit, the node a has to be in a different treelet than the node b. The word alignment pruning forces the node a' to be in the same treelet pair as the node a. Similarly, the node b' must be in the same treelet pair as the node a. Similarly, the order of the treelet pairs: based on the left



Figure 2.3: Problematic situations for word-alignment pruning. The dotted lines indicate the word alignment.

tree, the treelet containing a has to precede the treelet containing b but the right tree requires the reverse order. Therefore such a synchronous decomposition cannot exist.

Figure 2.3 (b) remains parsable under InternalsReflectWali: the derivation can proceed in two steps, we start with a treelet pair containing a and a' (and the son of a' as another internal node), and follow with a treelet pair containing b and b' (and the parent of b' as another internal node). With the more restrictive pruning of TreeletsReflectWali, there is no way to decompose the sentence. Every treelet containing the node a must include the node b at least as a frontier. The corresponding nodes a' and b' are however too distant to belong to a single treelet (even if b' were to be a frontier node).

An example of such situations in the data is shown in Figure 2.4. We see both the head switching (regardless an error in the automatic word alignment) as well as too distant nodes: The nodes "a", "jiný<sub>1</sub>", "and", "other" play the roles of the nodes a', b', a, b from Figure 2.3 (b), respectively. The only chance to construct the illustrated tree pair is to use a less restrictive pruning method and (asynchronously) deconstruct one side and construct the other always using treelet pairs where one of the treelets is null.



Figure 2.4: An example of a tree pair which is not covered by pruned set of rules.

Figure 2.4 illustrates one more problem. The synchronous decomposition has to start with a treelet pair rooted in the pair of roots of the trees. If all such treelet pairs are pruned (which happens in Figure 2.4), then the decomposition is impossible at all. This can be fixed by the addition of technical roots to all trees.

### 2.4.7 Experiments with Pruning Methods

Table 2.7 summarizes our experiments with various pruning methods and the corpus CzEng. With no pruning at all, only 419 sentences could have been processed before the fixed limit of 4 GB RAM was exhausted (about 13 million rules were created, giving us about 31 thousand rules per sentence). With the most restrictive pruning based on word alignment, i.e.

TreeletsReflectWali, 833 thousands sentences were processed before the memory was exhausted with about 37 rules per sentence. However, the strict pruning drastically reduced the coverage: only 43% of sentences had a synchronous parse using the extracted rules (see "Tree pairs reconstructed"). The most promising combination so far appears to be FrontierMapping-ReflectWaliBenevolent + InternalsReflectWali: the whole corpus can be processed and the rules still fit to the 4 GB limit, however, the coverage remains rather poor: only about 37% of sentences can be reconstructed.

		Tree pairs	Tree pairs
Pruning method	Rules extracted	processed	reconstructed
No pruning	12,993,863	419	419
TreeletsReflectWali	$30,\!977,\!163$	$833,\!133$	$354,\!264$
InternalsReflectWali	$16,\!475,\!748$	$7,\!183$	3,282
${\it FrontierMappingsReflectWaliStrict}$	$1,\!852,\!532$	$1,\!475$	$1,\!475$
$\label{eq:FrontierMappingsReflectWaliBenevolent} FrontierMappingsReflectWaliBenevolent$	$2,\!404,\!958$	$1,\!475$	1,475
Pruning of low frequency treelets	7,715,266	$1,\!007,\!305$	$63,\!350$
${\it FrontierMappingReflectWaliBenevolent}$			
+ InternalsReflectWali	68,725,228	$1,\!007,\!305$	$370,\!591$

Table 2.7: Comparison of various pruning methods.

### 2.5 Remarks on Perl Reimplementation

### 2.5.1 Motivation

The Perl implementation was developed in order to meet following goals:

- 1. To abridge the source code: The original Java implementation is a multi-purpose software, it is able to load several input formats, to use Prague Dependency Treebank data specifics and contains several obsolete configuration options. In contrast, the Perl implementation is kept as small as possible having the single functionality of training STSG.
- 2. To decrease memory requirements: all structure which appear many times during computation (e.g. treelets and rules) are represented by space-saving data structures with minimal overhead costs. Moreover Perl seems to have less memory consuming implementation of associative arrays (hashes).

Both of these goals were achieved. The source code of the whole program is only about 1000 lines long. The size of largest grammar that fits in the memory is greater than in the Java implementation. The comparison of the implementations is shown in Table 2.8.

Implementation	Rules extracted	Trees pairs processed
Java	$6,\!387,\!168$	297
Perl	$12,\!993,\!863$	419

Table 2.8: Comparison of Java and Perl implementation

### 2.5.2 Modules Documentation

The Perl implementation consists of five executables and three modules:

observe\_synchronous\_rules.pl: the "zeroth" iteration of observing and storing all (possibly pruned) synchronous rules in the training corpus. Probabilities estimated by maximum

likelihood are assigned to the rules. The initial scoring model used by first iteration of EM loop consists of these rule–probability pairs.

- em\_iteration.pl: an iteration of EM loop.
- filter\_sentences.pl: simple filter which omits trees containing a node with 6 or more children.
- treelet\_counts.pl: computes the number of treelet occurences in the training corpus. These counts are used for pruning of low frequency treelets as described in Section 2.4.5.
- transcode\_words\_to\_numbers.pl: filter which numbers lemmas, formems and functors / semposes in the input files in order to decrease size of a rule as described in Section 2.4.1.
- PSTSG\_Input.pm: module of subroutines for reading the input file and converting its lines to tree structures.
- **PSTSG\_Treelets.pm:** module of subroutines for extracting and serialization of treelets, treelet pairs and synchronous rules and subroutines for all kinds of pruning.

PSTSG\_Config.pm: module which reads both command-line and configuration-file options.

### 2.6 Future Plans

### 2.6.1 Using External Storage

We could either use a custom implementation to write observed rules directly to disk and use external sort to obtain their tree counts or we could simply tie Perl hashes to an external database using e.g. the module DB\_File.

### 2.6.2 Improving Pruning Methods and Combining with Backoff Models

All the experiments reported in Section 2.4.7 use the full model. While the pruning of low frequency single treelets prunes too many rules (coverage drops to 6% of sentences) in the full model, it could perform substantially better with less fine-grained backoff models.

We could also reduce the constraints of the alignment based methods, e.g. the Internals-ReflectWordAlignment could demand only one of the internals to be aligned correctly as most of the rules kept by this method are only one-internal ones.

Moreover, other pruning methods can be developed, e.g. we could demand a similar position (or distance from the root node) in the paired treelets.

### 2.6.3 Impact on TreeDecode

The transfer step (see Chapter 3) still uses heuristics to extract the dictionary of treelets. We will soon investigate the performance of the transfer step, if treelets are extracted and more importantly scored using the tree aligner.

### Chapter 3

## Tree Decoder: TreeDecode

In this chapter, we describe the implementation of the tree decoder, TREEDECODE. TREEDE-CODE was developed for Linux, primarily intended for the i686 and x86\_64 architectures, but we believe that without much effort, it should be portable to other platforms and architectures as well.

We first give the reader a general overview of TREEDECODE's functionality and then describe the important steps and their configuration in detail.

### **3.1** General Overview and Features

TREEDECODE runs *experiments* defined by the user on some given training data and input data. The result of an experiment is the translated text as well as an automatic estimate of output quality, if some reference translation was provided in the input.

Each experiment is fully described by its *configuration file*.

### 3.1.1 TreeDecode Operation

Based on a single configuration file, TREEDECODE performs the following tasks, see Figure 3.1 for a flow chart:

- 1. Load the configuration file.
- 2. Train and/or load translation and scoring models.
- 3. Optionally tune model weights in several MERT iterations:
  - (a) Load tuning input sentences with reference translations.
  - (b) Translate the sentences using current weights.
  - (c) Evaluate output *n*-best lists using BLEU or tree edit distance.
  - (d) Use MERT procedure to find new weights.
  - (e) Repeat the translation with new weights until locally optimal weights are found.
- 4. Load test input sentences (optionally with reference translations).
- 5. Translate input sentences; each sentence is translated in the following stages:
  - (a) Create translation options for each node in the source tree.
  - (b) Gradually expand partial hypotheses using translation options, covering the source tree top-to-bottom and constructing possible target trees synchronously.
  - (c) Extract *n*-best list of the highest-scored target trees.
  - (d) Optionally linearize the target trees with an external linearizer and rescore them using an n-gram LM.



Figure 3.1: Data flow in TREEDECODE.

6. Optionally evaluate output translations using BLEU (linearized output) or tree edit distance.

### 3.1.2 Parallelization and Caching

We use two complementary techniques to greatly speed up the run of TREEDECODE:

- Both training (collection of treelet counts) as well as translation of tuning and input sentences can be parallelized using Sun Grid Engine.
- Intermediate results are persistently cached (i.e. stored on the disk). For instance, if a model is created according to some configuration, both the exact configuration as well as the pathname to the model are stored in a disk cache. If any other TREEDECODE process needs to build a model based on the same configuration, the cached model is reused. File locking is used to force all TREEDECODE processes to wait for the single first worker, if the model is not finished yet.

The caching and parallelization fit nicely together. In parallel execution, there is a master process that uses SGE to launch slave processes. Slave processes can run on different computers so there is no way to share memory with them. It would also be greatly inefficient to send all model data to them using inter-process communication (e.g. a bi-directional pipe). Instead, we rely on a shared disk (NFS) and send only the configuration to the slaves. As the slaves try to construct all the models needed for the configuration, they find the models already constructed in the cache and all they have to do is to load them.

Data serialization techniques we use (the module pickle from Manarchive) are dependent on both the platform (i686 vs. x64) as well as the exact type definition of the serialized data term. To ensure compatibility between the master and the slaves, both functionality is implemented in the same executable file treedecode. When launched with some special arguments on the command line (never used by users directly), the code runs in one of so-called "slave modes", waiting for serialized input on stdin and serializing the output of the required predicate back to stdout. For each externalized predicate (i.e. the predicate that runs sometimes in a separate process), there is a special slave mode. Each of the supported slave modes thus corresponds to one of the externalizable predicates.

TREEDECODE in the main process tries to correctly handle various exception cases that can occur in slave processes. Some situations may still have remained untreated, which usually results in broken communication between the master and a slave. The master then complains with messages like "unable to unpickle functor lex". In such cases, the actual error is reported in the logfile of the respective slave process.

### 3.2 Users' Manual

### 3.2.1 Installation

TREEDECODE requires a Mercury compiler of the development version rotd-2007-05-05 or later. The latest stable version as of the writing of this document, 13.1, is *not* sufficient, as TREEDECODE uses some advanced features introduced into the compiler after the release.

1. Get the source codes and the demo input files from the following page:

http://ufal.mff.cuni.cz/euromatrix/

To simplify the installation process, treedecode-0.8.tgz package is self-contained.

2. Extract the archive:

tar xzf treedecode-0.8.tgz

3. Compile the contents:

```
cd treedecode-0.8/src make
```

### 3.2.2 Quick Run

To perform a simple experiment, run:

make demo-tecto

See the file Makefile for the exact command issued. The demo will uncompress a tiny sample training corpus, run an experiment using the configuration confs/basic\_tecto\_to\_tecto/conf (see Appendix B) and print where the log files are.

The final summary at the end of the standard error log file (the file err) indicates that the demo translated 10 sentences. The most frequently used model was to keep a node not translated ("unk"). The input trees had 14.4 nodes on average. Based on the tree edit distance metric, the agreement with the reference translation was about 45 % (or about 48 % if only node lemmas are considered):

```
Translated 10 sentences (0 had no translation) in 1.13s (0.1+-0.1s/sent, ...
Tropts used: unk:56 tr:52 wfw:30 wfwlemfunc:3
For the translated: avg input words: 14.4+-6.2, output words 14.4+-5.8, ...
10 sents: exact=45.45%, wprec 24.3%, wrec 20.0% (0 unscored), f0t_lemma=47.65%...
```

The outputs are stored in the **out** file. Each output sentence is on a separate line and the output tree structure is rendered using square brackets and the hash symbol (#) to indicate the position of the father node among its sons, e.g. (line breaks and indentation added for clarity, listing of node attributes abbreviated):

```
[--root--
#
[noon
   [t_lemma|další|||functor|RSTR|||gram/sempos|adj.denot... # ]
   [balkan # ]
   [t_lemma|vysoký|||functor|RSTR|||gram/sempos|adj.denot... # ]
#
]
```

The auxiliary root is always labelled --root--. Nodes without a translation usually consist of one output factor only (e.g. noon), other nodes are printed with all output factors.

For debugging and illustration purposes, the input tree, best hypothesis and the reference translations of each sentence are also produced as a scalable vector graphics (SVG) file, e.g. clip1.svg in our example. The SVG output is not only easier to interpret, but it also indicates the decomposition of the input tree and the corresponding composition of target-side treelets.

### 3.2.3 Configuring an Experiment

Each experiment is fully described by a structured configuration file. In the following pages, we shall describe the syntax of the configuration file formally so that there are no ambiguities. For the meaning of some options, the reader is referred to Bojar and Čmejrek (2007) for mathematical and algorithmic description of tree transduction.

TREEDECODE is implemented in Mercury and uses its syntax (which is vaguely similar to that of Prolog) for the configuration file. When loading the file, TREEDECODE looks for the term config/13 and tries to interpret it using the rules below. No further checking is done— if the term is successfully parsed, it is deemed sane and the process described in Section 3.1 proceeds to step 2.

TREEDECODE allows users to declare variables in the configuration file. These are provided only as a means to reduce redundancy and improve clarity and are not visible anywhere later in the process. At the end of the loading of the configuration, they are substituted into the main term. The file may contain an arbitrary number of variable definitions, but their names must not clash and each variable must be used after it is defined.

Note that in the present version, we include some runtime environment settings in configuration files as well. We plan to separate all settings that do not influence the output into special files so that the configuration would describe only the experiment regardless of its actual execution.

#### Formal Syntax of a Configuration File

In the following, we will use this font to denote terminals (i.e. strings that occur verbatim in the configuration file), this font to denote non-terminals (i.e. types, in fact) and this font to denote built-in types. Also, [X] will denote a list whose elements are X's (be it terminals, non-terminals, built-in types or combinations of these). Similarly as in Prolog, having e.g. three elements a, b and c, the list that contains these elements in the given order is written as [a,b,c]. [] is the lexical representation of an empty list.

The built-in types are integer, float and string. Their lexical form is the same as that of the corresponding types in the C language.

Configuration files may also contain comments of two kinds: Prolog-style, i.e. from the character "%" to the end of the line, and C-style, i.e. from "/\*" to "\*/". White space (other than that in string tokens) is ignored.

The formal syntax of the configuration file (represented here by the non-terminal *ConfigFile*) after the removal of comments and non-significant whitespace characters is the following:

• ConfigFile → VariableAssignments ConfigTerm

Variable assignments are defined as follows:

- $VariableAssignments \rightarrow \lambda$
- $VariableAssignments \rightarrow VariableName = AssignableTerm$ . VariableAssignments
  - VariableName  $\rightarrow$  string

variable name must conform Prolog-like notation, i.e. an uppercase letter optionally followed by alphanumeric characters or underscores.

Only a subset of all terms is allowed to be assigned to a variable:

- $AssignableTerm \rightarrow \texttt{factors\_config}(FactorsConfig)$ for the definition of FactorsConfig, see page 26.
- $AssignableTerm \rightarrow \texttt{factors\_to\_use}(FactorsToUse)$ for the definition of FactorsToUse, see page 28.
- Assignable Term  $\rightarrow$  corpus (Corpus) for the definition of Corpus, see page 26.
- Assignable Term  $\rightarrow$  limits (TreeletSizeLimit) for the definition of TreeletSizeLimit, see page 24.
- $AssignableTerm \rightarrow dirconf(DirConf)$ for the definition of DirConf, see page 27.
- $AssignableTerm \rightarrow fftable(FFConfig)$ for the definition of *FFConfig*, see page 28.
- Assignable Term  $\rightarrow$  int(integer) an integer value.

After variable expansion, the main configuration term has the following syntax:

• ConfigTerm → config(OutputType, TreeletSizeLimit, StackSizeLimit, Tweaks, Weights, MERT, TestDataSource, FactorsConfig, Evaluation, DirConf, RecombEquality, TrgConf, OutputCollectors).

Omissions of some of the arguments or substitution of variables at wrong places results in a type error.

We shall now describe each of the items in detail.

Output Type determines the sort of the result of the whole process. Valid options are

- *OutputType* → produce\_string the output is a text (string).
- *OutputType* → produce\_tree the output is a tree.
- **TreeletSizeLimit** are the limits for treelet size. In the main config term, the limits are applied to input trees before translation. In the individual models, the limits are applied to both source and target trees when extracting treelets.

Treelets exceeding these size constraints are discarded. In some cases, this can result in sentences with no translation at all (e.g. due to excessive branching).

- TreeletSizeLimit → limits(MaxInternal, MaxFrontier, MaxBranching)
  - MaxInternal  $\rightarrow$  integer maximum number of internal nodes.
  - MaxFrontier  $\rightarrow$  integer maximum number of frontier nodes.
  - $MaxBranching \rightarrow integer$ maximum branching factor of the treelet, i.e. the maximum number of children of each node.

*StackSizeLimit* is an integer that specifies the maximum length of the beam-search stack.

•  $StackSizeLimit \rightarrow integer$ 

**Tweaks** were introduced to simplify adding new options to the decoder without the need to change the main config type. This allows to keep old config files unchanged, even if a new tweak is added. The old config file will be launched with the tweak set to a default value.

Moreover, there is a quick way of adding a new command-line option to set the value of a particular tweak. So in a way, tweaks can be seen as a direct reflection of some command-line options in the model.

The experience so far suggests that there are rather only few situations where these nonstructured configuration options are useful. In general, it is better to keep the flags close to the modules that use them.

Tweaks are defined as an associative list, i.e. a list of key-value pairs:

•  $Tweaks \rightarrow [TweakName-TweakValue]$ 

 $- TweakName \rightarrow \texttt{tropt_limit}$  limits the number of translation options

- TweakValue  $\rightarrow$  i(integer)
- **Weights** is a list of floating-point numbers that determine the weights of the scorers. If the list is empty, the word and phrase penalty scorers are set to -1 and the rest to 1. If the number of weights does not match the number of lambdas introduced by models in the configuration, TREEDECODE aborts.
  - $Weights \rightarrow [float]$
- $MERT\,$  configures the settings of Minimum Error Rate Training, the iterated weight optimization procedure.
  - *MERT* → no\_mert do not use MERT, translate with the specified (or default) weights.
  - *MERT* → mert(*NBestList*, *RandomStarts*, *MERTDevSource*, *FactorsConfig*, *MERTDevRefs*) perform the MERT training as described in Och (2003), using BLEU as the objective function.
    - NBestList  $\rightarrow$  integer the length of the *n*-best list to use.
    - RandomStarts  $\rightarrow$  integer the number of random starts.
    - $MERTDevSource \rightarrow string$ source file to be used for the training process. The fourth argument of the mert term specifies the interpretation of the factors given in the file (for definition of *FactorsConfig*, see below).
    - $MERTDevRefs \rightarrow [string]$ reference files for BLEU.
  - $MERT \rightarrow try\_some([Probe], string, FactorsConfig, Evaluation)$

this option is useful for more or less manual exploration of the weights vector space. As opposed to MERT, which performs a search through the vector space, this method finds the best vector of a given fixed set of vectors. The evaluation method to use is determined by *Evaluation* (see page 27 for its definition).

*Probes* provide generic methods for generation of the weight vectors.

-  $Probe \rightarrow \texttt{all_equal}$ 

vector with all scorers' weights set to 1.

- Probe  $\rightarrow$  wordpenalties\_negative vector with all scorers' weights set to 1, except scorers for source and target word penalties, which are set to -1.
- Probe → one\_vs\_rest(float, float)
   all vectors with one component set to the value of the first argument and the remaining to the value of the second argument.
- Probe → one\_vs\_rest\_vs\_wordpenalties(float, float, float)
   all vectors with word penalty scorers weitghts set to the value of the third argument, one of the remaining components set to the value of the first argument and the remaining components set to the value of the second argument.

TrainDataSource defines the source of the training data.

- *TrainDataSource* → stdin the data will be read from the standard input.
- $TrainDataSource \rightarrow file(string)$ the data will be read from the file specified in the argument.

Corpus is used to specify a training parallel corpus.

- Corpus → corpus(string, FactorsConfig, FactorsConfig) TREEDECODE's view of a bilingual corpus determined by the corpus file name (the first argument), and the interpretation of source and target language word factors (the second and third argument, respectively).
- **FactorsConfig** provides mapping from factors in the input data to factors in the internal structures. In the main configuration term, the *FactorsConfig* is concerned with the interpretation of input test file, in the *Corpus* term, *FactorsConfig* describes the corpus file.
  - FactorsConfig  $\rightarrow$  factors\_config(GovFactor, WordFactor, FstateFactor) GovFactor says which factor is to be used as the index of each node's head (the root of the tree has index 0).
    - $GovFactor \rightarrow verbatim(integer)$ use the factor at the index given by the argument.
    - GovFactor → verbatim\_rehanging\_final\_punct\_to\_root(integer, integer)
       fix parses parsed by the Collins parser: if the word form (pointed to by the first argument) of the last word is '.' or '?', set the value to 0, otherwise use the value of the factor given by the second argument.
    - $GovFactor \rightarrow linear\_tree$ construct a linear tree, i.e. the head is always the previous word.

WordFactor defines the factors for internal nodes.

WordFactor → wordfactor([string-FactorConstructor])
 an associative list that assigns names (the first element of each pair) to factor constructors. (For the definition of FactorConstructor, see below.)

*FstateFactor* defines the factor for frontier state whenever a node is used as a frontier.

- FstateFactor  $\rightarrow$  fstatefactor(FactorConstructor)

FactorConstructor describes what factors from the input data to pick and bind together.

• FactorConstructor → unit the factor does not carry any information.

- FactorConstructor  $\rightarrow$  verbatim(integer) take the exact value of the given factor (as specified by the index).
- FactorConstructor → cs\_agr\_1(integer, integer, integer)
   constructor for modelling agreements (for Czech) based on (Bojar, 2007) the arguments refer to (in the following order) the word form, lemma and tag factors.
- *FactorConstructor* → en\_fm\_phr\_pos(integer, integer, integer) constructor for joining the word form, phrase and tag factors; again, the arguments correspond to the factor indices.
- FactorConstructor → complex([string-integer]) an associative list used to construct an atomic factor by "gluing together" an arbitrary combination of input factors. The first element of each string-int pair allows yout to provide a name of the component of the complex factor. The second element is the index of the input factor.

**Evaluation** configures the evaluation of a translation output.

- Evaluation → no\_evaluation
   do not perform any evaluation of the output.
- Evaluation → evaluate\_bleu([string])
   for linearized output, evaluate the quality of translation using BLEU. The argument defines the names of reference files to use.
- Evaluation → evaluate\_trees(TreeEvaluation, LinearizedEvaluation) for non-linearized output, evaluate the quality of translation using a tree edit distance metric (see (Shasha and Zhang, 1997)) and/or using BLEU after linearization.
  - $TreeEvaluation \rightarrow no\_tree\_evaluation$ do not score the output trees.
  - $TreeEvaluation \rightarrow \texttt{treedist(string, } FactorsConfig, [EvalFactor])$ evaluate the tree distance of the output to the reference translations.
    - \*  $EvalFactor \rightarrow \texttt{all\_factors}$ use all factors.
    - \*  $EvalFactor \rightarrow single_factor(integer)$ use a single factor, the argument being its index. Note that although *WordFactor* is an associative list that assigns each internal node factor a name, for implementation reasons we index the factors by the order they appeared in the list.
    - \*  $EvalFactor \rightarrow section_of_complex_factor(integer, string)$ pick a single section from a complex factor. As above, the word factor is referenced to by its order of appearence in the list instead of its name.
  - $LinearizedEvaluation \rightarrow no\_linearized\_evaluation$ do not perform evaluation of linearized output.
  - LinearizedEvaluation  $\rightarrow$  linearize\_and\_bleu(string, [string]) linearize output trees and score the sequences using BLEU. The first argument is the full path to an external linearizer, the second is a list of reference files for the BLEU metric.
- DirConf is the runtime environment configuration, declaratively not related to translation results.
  - DirConf → dirconf(CacheDesc, ModelDirName, LocalTemp, ScratchSpace) CacheDesc configures the generic cache for cached predicate calls:

-  $CacheDesc \rightarrow cache\_desc(string, CacheSubdirs)$ 

The first argument defines the path to the cache directory.

- \*  $CacheSubdirs \rightarrow no$ do not use subdirectories for the cache.
- \*  $CacheSubdirs \rightarrow yes(integer)$ use the given number of subdirectories for the cache. This reduces the number of files per directory and can thus improve the performance of e.g. NFS file servers.

ModelDirName is the path to the directory in which TREEDECODE will store finished models:

 $- \ \mathit{ModelDirName} \rightarrow \mathsf{string}$ 

LocalTemp configures the usage of a local temporary directory for models that use the module **biprobdb** for distributed estimation of probabilities (newer models use **hugemap** instead):

- LocalTemp  $\rightarrow$  local\_temp\_directory(string) write intermediate files to a local temporary directory first and move them to the models directory after they are finished. In cluster environments, this can help reduce the network load.
- LocalTemp  $\rightarrow$  write\_directly

write intermediate files directly to the models directory.

*ScratchSpace* sets up the working directories of other implementation mechanisms, such as journals used by the module hugemap.

- $ScratchSpace \rightarrow fixed\_scratch\_space(string)$ use the fixed path given in the argument for the scratch space.
- ScratchSpace  $\rightarrow$  guess\_scratch\_space\_from\_hostname guess the scratch space from the host name. Note that the way the scratch space name is guessed is specific to ÚFAL network.

**RecombEquality** defines the equivalence class of partial hypotheses for the purpose of reducing unnecessary work by expanding similar hypotheses only once per the entire class.

- *RecombEquality* → opened\_positions\_equal hypotheses are recombined if the sets of currently pending frontier nodes including state labels are equal.
- $RecombEquality \rightarrow \text{opened_positions\_and\_some_father\_attrs\_equal(}FactorsConfig, FactorsToUse)$

hypotheses are recombined if the sets of pending frontier nodes including state labels and the selected factors of their parent (internal) nodes match.

- FactorsToUse  $\rightarrow$  [string]

the list of names of factors that are to be used.

**FFConfig** specifies the configuration of a probabilistic mapping from source to target frontier states. This mapping is used in various translation option generators (see below).

• FFConfig → not\_inited(config(string, DirConf, Corpus, FstateFstateLimit, SrcParentFactors, TgtParentFactors, BackoffToUnconditioned, DefaultFstate))

the first argument defines scorer prefix for the generator, *DirConf* is the environment configuration, *Corpus* selects the source corpus (for the definition of *Corpus*, see page 26).

FstateFstateLimit is an integer specifying how many target frontiers states should be considered for a source frontier state.

SrcParentFactors and TgtParentFactors specify a subset of the factors of source and target governing nodes to condition the mapping on.

*BackoffToUnconditioned* optionally allows to use parent-unconditioned frontier state mapping if the conditioned situation was never seen. Valid options are:

- BackoffToUnconditioned  $\rightarrow$  backoff\_to\_unconditioned
- $\textit{ BackoffToUnconditioned} \rightarrow \texttt{dont\_backoff\_to\_unconditioned}$

*DefaultFstate* is used as an ultimate backoff for unknown source frontier states. Valid options are:

- DefaultFstate  $\rightarrow$  no
  - don't allow ultimate backoff, just fail to translate the frontier state.
- $DefaultFstate \rightarrow yes(string)$  use the specified string as the frontier state if nothing better available.

TrgConf configures the sequence of translation option generators and their settings.

- $TrgConf \rightarrow trg(TrProviderConfig)$ register a translation option generator. For the definition of TrProviderConfig, see below.
- TrgConf → first\_of([TrgConf]) given a list of translation option generators, pick the first one that succeeds.
- TrgConf → merge\_all([TrgConf])
   try all of the listed translation option generators and merge the results.
   Note that the current implementation of merge\_all suffers two problems: (1) if several generators produce the same output, the translation option is considered several times in the main search, and (2) each translation option is scored only by the single generator that produced it, other scores are set to zero (e.i. probability of 1). Ideally, each option should be scored by all translation option generators.
- $TrProviderConfig \rightarrow t\_preserve\_all(PreserveAll)$ the basic translation output generator (see Section 3.5.1 in Bojar and Čmejrek (2007)).
  - PreserveAll → config(string, DirConf, Corpus, FactorsToUse, TreeletSizeLimits, Generation, Filter, Matching, DropAdjuncts, TreeletConversion)
     the first argument defines scorer prefix for the generator, DirConf is the environment configuration, Corpus selects the source corpus (for the definition of Corpus, see page 26) and FactorsToUse source factors to be used. TreeletSizeLimits are the limits of the size of treelets generated during the training phase. Generation determines the way how the pairs of source-target language treelets are generated:
    - \* Generation  $\rightarrow$  ignore\_ali ignore alignment altogether, i.e. generate all possible combinations.
    - \* Generation  $\rightarrow$  linked\_roots roots of both treelets must be aligned.
    - \* Generation  $\rightarrow$  linked\_roots\_covered\_internals roots must be in alignment and for each internal, if aligned at all, the alignment link must remain within the treelet pair.
    - \* Generation → climb\_linked\_roots\_covered\_internals is an extension of linked\_roots\_covered\_internals that allows head switching: the roots have to be aligned, but not necessarily to each other. Alignment to any internal node in the other treelet is sufficient.

*Filter* defines the constraints for exclusion of some generated treelets.

\*  $Filter \rightarrow no\_filter$ 

do not filter out anything.

Matching determines how to generate the links between frontier nodes.

- \*  $Matching \rightarrow any$
- generate all possible combinations.
- \*  $Matching \rightarrow none$ do not link together any frontiers.
- \*  $Matching \rightarrow \texttt{ali_compatible}$

generate all combinations that are compatible with the alignment.

DropAdjuncts defines the condition for dropping adjuncts from the generated treelet.

- \*  $DropAdjuncts \rightarrow \text{keep-all-adjs}$  do not drop any nodes.
- \*  $DropAdjuncts \rightarrow drop\_adjs\_where\_src\_fstate\_is([string])$

assume all frontier nodes with one of the labels given in the argument are adjuncts and drop them from the treelet. During the translation, adjuncts are translated independently and using the same distribution for all treelets.

TreeletConversion specifies the finalizing conversion of the left-side (i.e. source) treelets.

- \* TreeletConversion  $\rightarrow$  no\_conversion do not perform any conversion, keep the treelets as they were.
- \* TreeletConversion  $\rightarrow$  sort\_treelet sort the frontier nodes of source treelets alphabetically by their labels. This reduces data sparseness, but also decreases accuracy because source word order is ignored.
- TrProviderConfig → t\_word\_for\_word\_factored(WordForWordFactored) factored word-for-word translation.

Note that this translation option generator can be used also for non-factored word-for-word translation and for keeping untranslated words as described in Sections 3.5.2 and 3.5.4 of Bojar and Čmejrek (2007).

WordForWordFactored → config(string, DirConf, Corpus, [Step], TargetFstateFrom, FFConfig, FactStackLimit)

the first argument defines scorer prefix for the generator, *DirConf* is the environment configuration, *Corpus* selects the source corpus (for the definition of *Corpus*, see page 26).

- \*  $Step \rightarrow \texttt{translate}(FactorsToUse, FactorsToUse, FactorsToUse)$ based on the value of some FactorsToUse (arg. 1) of the source parent node and some FactorsToUse (arg. 2) of the source internal node, predict or check the value of some FactorsToUse of the target internal node.
- \* Step  $\rightarrow$  generate(FactorsToUse, FactorsToUse) based on FactorsToUse (arg. 1) of the target node, predict or check other FactorsToUse of the target node.
- \* Step → copy(FactorsToUse, FactorsToUse)
  copy some source FactorsToUse (arg. 1) to some target FactorsToUse (arg. 2) verbatim, i.e. do not translate the values. The two lists of factors must be of the same length.
- \* Step  $\rightarrow$  delay(Factors To Use) mark some target factors as "delayed", i.e. do not generate their values until the whole tree is constructed in the main search. Note that the current implementation does not support the final filling of delayed factors yet.

- \*  $TargetFstateFrom \rightarrow target_fstate_from_factor(string)$ the root state of the translation option (i.e. where can the translation option attach to) is taken from the given target factor.
- \*  $TargetFstateFrom \rightarrow \texttt{target_fstate_probabilistically}$ the root state of the translation option is generated probabilistically based on the source root state using the mapping specified by *FFConfig*.
- \* FFConfig (see page 28) is used to map source frontier states to target frontier states.
- \*  $FactStackLimit \rightarrow integer$ size of stack for generating partial hypotheses in this step.
- TrProviderConfig → convert\_number\_encs(FFConfig)
   convert a number written in an English style to the Czech style. E.g. "4.5" is converted to "4,5". FFConfig is used to map frontier states. This rule-based translation option generator is not suitable for factored output.
- **OutputCollectors** collect and optionally rescore generated output sentences, which is done through an automatically assigned lambda.
  - OutputCollectors → oc\_segments(config(not\_inited(LMConf)) output collector for serialized (text) output. The items are optionally scored using a language model.
    - $LMConf \rightarrow no_lm$

do not use any language model.

- LMConf  $\rightarrow$  lmconf (LMOrder, LMFile, LMNgramFutureCost) use an n-gram language model:
  - \*  $LMOrder \rightarrow integer$ the order of the model (n).
  - \*  $LMFile \rightarrow string$ path to the file containing the model.
  - \*  $LMNgramFutureCost \rightarrow \text{float}$ determines how much is an unfinished hypothesis charged for all *n*-grams that still have to be scored.
- OutputCollectors → oc\_tree(OutputType, [BinodeLM], LinearizedRescoring) output collector for trees. The items are scored using a bi-node language model and optionally by a language model after linearization.
  - $BinodeLM \rightarrow not\_inited(config(string, DirConf, Corpus, FactorsToUse, FactorsToUse))$

use a bi-node language model.

- LinearizedRescoring  $\rightarrow$  no\_linearized\_rescoring
- LinearizedRescoring → linearized\_rescoring(Linearizer, [not\_inited(LMConf)]) do rescore the tree after linearization using language models (the second argument).
  - \* Linearizer  $\rightarrow$  string the path to the external linearizer.

### 3.2.4 Running TreeDecode

TREEDECODE is influenced by several command-line options and environment variables.

### **Command-line Options**

TREEDECODE recognizes the following command-line options:

• -h or --help show a help message.

The following options influence the way TREEDECODE is run.

- -c FILE or --config FILE use FILE as the configuration file. This option is required.
- --chdir DIR

change the working directory to DIR before doing anything. This is useful if the config file specifies paths relative to a different origin.

• --chdir-back

after finishing all work, change the working directory back to the working directory from which TREEDECODE was run. This is useful for profiling the program because profiler logs are always written to the current working directory after the program finished.

The following options override of the settings from the experiment's configuration file, see Section 3.2.3.

- -i FILE or --input FILE use FILE as the input file
- -w WEIGHTS or --weights WEIGHTS define weights for the scorers. WEIGHTS should be a comma-delimited list of floating point numbers. The number and order of the weights is implied by the models used in the configuration file and the list of weights specified using --weights has to conform to that.
- -s LIMIT or --stack-limit LIMIT set the beam-search stack size to LIMIT.
- -t LIMIT or --tropt-limit LIMIT set the maximum number of translation options to LIMIT.

### Environment Variables for Memory and Grid Usage

- KILL\_LIMIT=80.0 specifies TREEDECODE's memory resource limit as set by the the setrlimit system call. It is a percentage of the computer's memory TREEDECODE is allowed to use (e.g. 80%). If the limit is exceeded, the kernel aborts the process with a segfault. (A normal segfault is not distinguishable from this kernel-assisted suicide.)
- SGEJOBS\_TRAIN=Num, SGEJOBS\_TEST=Num, GRIDJOBS=Num specify the number of jobs submitted to the Sun Grid Engine. When set to 0, the Grid Engine is not used at all. While GRIDJOBS is used when collecting treelet counts in training, SGEJOBS\_TEST specifies the number of parallel jobs when translating. SGEJOBS\_TRAIN is becoming obsolete but it is still used in some older models in the training phase.

### Environment Variables for Debugging of the Models

The following environment variables can be used to obtain additional debugging outputs. Most of the variables function as simple switches, if set to a value (e.g. "yes"), the respective debugging output is printed to standard error output.

Carefully check the size of the logs created to avoid excessive load of your network or file server.

- JUSTSENT= $Num_1, Num_2, \ldots$  restricts the translation only to the test sentences  $Num_1, Num_2, \ldots$ and disables both caching and parallel execution of the translation. Useful when tracking down a problem occurring at a particular sentence.
- LOADTREE=yes and line\_to\_alitrees=yes print detailed information about every line read from the training or evaluation corpora. Beware, verbosely dumping your training corpus is likely to ruin your network or disk capacity.
- TROPT=yes prints a summary of translation options generated and kept at every node.
- **TROPTDETAIL=yes** prints detailed information about every translation option considered and generated, regardless if it eventually survives in the translation options stack.
- DUMPSTACKSIZES=yes prints the number of partial hypotheses in each stack after every iteration.
- DUMPSTACK=yes prints a summary of all partial hypotheses in a stack that are about to be processed.
- EXPAND=yes prints each hypothesis that is about to be expanded.
- EXPANDNEW=yes prints also the output of each hypothesis expansion.

**RECOMB**=yes prints all hypothesis pairs considered for recombination.

TRACEBACK=H30H,40,... allows to specify a set of partial hypotheses you wish to closely analyze. Once a hypothesis with one of the required ids is created, the history of the hypothesis is printed (i.e. the sequence of gradual expansions). Hypotheses can be specified as integers or as integers marked with H...H.

Search for the text "Traceback of" in the logfile.

TRACESONS=H1H,H30H,40... allows to specify a set of partial hypotheses you wish to closely analyze. Whenever one of the selected hypotheses is successfully expanded to create a new son, the new son is printed. Hypotheses can be specified as integers or as integers marked with H...H.

Search for the text "Son of" in the logfile.

- GETNBEST=yes prints detailed information from the backward search through the stacks when generating n-best lists.
- **RESCORE=yes** prints hypotheses and scores before and after the final linearized rescoring.
- SVGLOG=filename%.svg saves the source, hypothesis and reference trees for sentence *i* to file filename*i*.svg in SVG format.
- SVGLOGCOMMENTS=reffile.txt, commentsfile.txt augment the SVG logs generated by SVGLOG with plaintext comments from all specified files. Please note that all the files have contain exactly as many lines as there are input sentences.
- NOCACHE=yes completely blocks the disk cache. All intermediate results including training of models will be recomputed.

### Environment Variables for Debugging of the Decoder

The following environment variables are useful rather for developers of TREEDECODE.

STACK=yes print information about inserting and pruning the stack of translation options.

BACKOFF=yes prints detailed logs on when each of the back-off models was used during translation options generation.

- BIGEN, BIGENFILTER, and GENLINKEDCOVERED=yes print information about observed and filtered treelet pairs and treelet pairs generated using the method linked\_roots\_covered\_internals.
- t\_preserve\_all, t\_word\_for\_word\_factored, t\_drop\_fr\_reins, and binodelm=yes print detailed information from the corresponding model. Please note that each of the models provides different type of debugging output.
- biprobdb\_save\_plaintext=yes save the database also in plaintext format, in addition to the standard tinycdb database file.

### 3.2.5 Troubleshooting

Here is a short list of common problems with TREEDECODE.

**Outdated or incompatible cache files.** TREEDECODE uses a binary format to cache various intermediate results on disk. If you compile TREEDECODE on various platforms or modify cached data structures, existing cache files may become incompatible with the TREEDECODE executable. TREEDECODE may complain about the problem when deserializing the files, e.g.:

cached: Checking the cache or waiting for others: "/home/.../cache/..."
cached: Loading cached /home/.../cache/c000.782
Uncaught Mercury exception:
Software Error: nonsense, are your caches fresh?
Stack dump not available in this grade.

To fix the problem, just delete at least one of the relevant cache files, e.g. c000.782 in our example. TREEDECODE will re-create the models as needed.

Sometimes you may wish to delete the whole cache directory. Depending on your configuration, TREEDECODE can use different cache directories for different purposes, so you may need to delete several directories.

In very rare cases, the difference in binary representation might be so subtle that TREEDE-CODE will not notice (but the results will be wrong) or it will misinterpret the data and start allocating insane chunks of memory. In general, trust your configuration files and delete the caches if in doubt. If the problem persists, you have found a bug.

Not enough memory to finish the experiment. The training process is memory-demanding and Mercury's current garbage collector doesn't help very much—it is not uncommon for TREEDECODE to abort after exceeding the memory limits imposed by the environment variable KILL\_LIMIT.

If this happens, try increasing KILL\_LIMIT and re-running the experiment. Sometimes even a fresh run with the same KILL\_LIMIT may help because the previous predicate calls will be cached and not recomputed, greatly reducing the memory demands of the process.

### **3.3** Input Data Formats

TREEDECODE reads data in the following formats:

Word-aligned bitext is similar to that described in Section 2.1.3 at page 8. It is a plain text format whose rows consist of three columns, each separated by a tab:

- source language text
- target language text
- word alignment

Words in the source and target language columns are separated by a single space and may consist of factors that are separated by a '|' character.

The word alignment is defined by a list of edges between source and target language words, with the first word in each column having index 0.

Translation input corresponds to the "source language text" column as defined above.

**Reference files** correspond to the "target language text" column in the word-aligned bitext above—for BLEU, words do not consist of factors, for tree-distance based metrics, factors are required.

### 3.4 Experimental Loop

TREEDECODE has been implemented to manage the whole process of running and evaluating an experiment. In the main TREEDECODE Makefile, we provide some basic functionality to run many experiments and compare the performance. This generally speeds up the loop of designing an experiment, running it (which can take hours to days of CPU time), evaluating it and modifying it to achieve further improvements.

Source configuration files are stored in the confs/ subdirectory, with filenames ending with ".conf".

### 3.4.1 Starting an Experiment

To start an experiment based on a config file confs/CONFIG-NAME.conf, issue the following command:

```
make CONFIG-NAME.run
    # this will start the experiment in the background
make CONFIG-NAME.runhere
    # this will start the experiment in the console using 'less' to paginate
    # the log
```

Each experiment is launched in a fresh time-stamped directory to avoid any potencial clash of experiments executed in parallel. We call these time-stamped directories "runs" and store them in the **runs/** subdirectory.

For a full reference and to be absolutely sure which configuration file and which version of TREEDECODE was used for a particular run, a copy of the config file is saved to runs/ $\langle run \rangle$ /conf and a copy of the executable is saved to runs/ $\langle run \rangle$ /exe. Along with the executable and the configuration, there are standard error (err) and standard output (out) from the run. Later, you may not only wish to examine the configuration or log files, but you may also restart the experiment simply by issuing the following commands:

```
cd runs/THE-RUN-TO-RE-RUN
./exe -c conf > NEW-out 2> NEW-err
```

Other advantages of this strict separation of individual runs are that log files from slave jobs submitted to Grid Engine are stored together with the main log file of the experiment and also that e.g. profiler files won't get overwritten and several profiling experiments can be run in parallel.

### 3.4.2 Working Directory of an Experiment

As indicated in the previous example, a local copy of TREEDECODE is launched *in* the directory of the run. All paths to corpora or caches in the config file have to be specified either absolute or relative to the directory of the run.

For instance, if all experiments should access the same cache directories, the path to the cache should start with ../../ to leave the run and runs/ directories first.

### 3.4.3 Cloning an Experiment

The whole experimental loop typically consists of the following steps:

- 1. Create or modify a config file, e.g. confs/TEST.conf.
- 2. Start an experiment: make TEST.run.
- 3. Repeat.

After several iterations we may wish to return to an older version of the configuration. Similarly, after several related experiments have finished, we may wish to use the configuration of the best performing one for further improvements. The script cloneconf simplifies the copying of the config from a run directory back to confs/:

```
make TEST.run
# now watch runs/TEST-TIMESTAMP-1/err
# modify confs/TEST.conf, start other experiments
./cloneconf runs/TEST-TIMESTAMP-1/err
# the configuration in confs/TEST.conf is now back at the state
# used for runs/TEST-TIMESTAMP-1, modify or just rerun the experiment.
```

#### 3.4.4 Giving Experiments Symbolic Names

After having launched several experiments, the **run** directory can become cramped. If an experiment has already finished, you may freely rename the experiment directory.

To provide a simple means of labelling experiments when starting them, you may use the environment variable NAME:

NAME=extra-info make CONF.run

```
# this will run an experiment using the configuration from confs/CONF.conf
# and store it in runs/CONF.extra-info.20080811-18...
```

### 3.4.5 Collecting Results from Many Experiments

Typing make bleu will collect translation quality estimates from the err files from all runs to a single table, saved to the file bleu.

### 3.5 Source Code Overview

TREEDECODE is implemented in Mercury (Somogyi et al., 1995) with some functionality provided by additional C/C++ libraries. The whole system is compiled to a single executable file.

### 3.5.1 Libraries Used

TREEDECODE releases have a flat directory structure with all source files in a single directory. During the development, individual libraries are kept separate:

- **Manarchive**<sup>1</sup> is an archive of various Mercury predicates and functions for general use. Manarchive extends the standard Mercury library with functionality like fast and space-efficient serialization (pickling), file locking, reporting and reacting to current memory usage. Many routines in Manarchive were improved during the development of TREEDECODE.
- **Obomerclib** is a private collection of Mercury modules for accomplishing various tasks developed mainly by Ondřej Bojar. Once the interface and implementation of a module becomes relatively stable, the module is publicly released in Manarchive. Obomerclib contains modules for e.g. generic regular expressions and finite-state automata, limited support of Unicode, binding to zlib for direct access to gzip-compressed data, binding to the statistical software R, launching and communicating with external processes, persistent caching of predicate outputs (module cached), high-level API for drawing SVG pictures, and various versions of "externalization", i.e. the option to run a Mercury predicate in a separate process, including the support to run many of such externalized predicates in parallel on a multi-processor machine or on a Sun Grid Engine<sup>2</sup>
- **IRSTLM (Federico and Cettolo, 2007)** is an open-source n-gram language modelling in C/C++. TREEDECODE implements a Mercury binding to the library.
- **MERT** is Philipp Koehn's C/C++ implementation of minimum error rate training by Och (2003), as made publicly available in the Moses decoder. We include Philipp's code in TREEDECODE and implement a Mercury binding to it.

### 3.5.2 Description of Core TreeDecode Modules

The following modules constitute the core of TREEDECODE:

treedecode.m is the main module of TREEDECODE. It implements the main/2 predicate, loops through input sentences, MERT loop for weights optimization, as well as the core part of the search: gradual expansion of partial hypotheses, hypothesis recombination and stack management, *n*-best list extraction, and *n*-gram rescoring of linearized hypotheses.

Upon startup, **treedecode** checks the command line arguments. If they indicate a request for a slave mode, **treedecode** just runs the respective externalized predicate and exits afterwards. Otherwise, the usual ("master") code is performed.

core\_defs.m defines some widely-used data types. E.g., a "word" is an array of "factors"
 (strings), while a "partial\_word" might still have some factors not filled. Finally a "re stricted\_word" (defined in the module restrict.m) is a word restricted to a subset of
 factors.

Apart from the basic definitions, various global defaults are stored in core\_defs.m, e.g. how much memory should be used for in-memory caching when collecting treelet counts in the training phase.

- config.m defines the core of configuration data type, loads the configuration file, parses commandline options and incorporates them to the configuration.
- tropt.m defines and handles translation options. In other words, it is responsible for initialization (and training) of all models used for translation option generation as specified in the configuration. The module tropt.m also implements the first stage in a sentence translation, the creation of all possible translation options using these models.

<sup>&</sup>lt;sup>1</sup>http://manarchive.sf.net/

<sup>&</sup>lt;sup>2</sup>http://gridengine.sunsource.net/

t\_generic.m defines an abstract interface for all translation option generators that are trained by collecting counts in a corpus. The module t\_generic.m simplifies as generalizes the access to the corpus and counting in parallel on SGE.

Various translation option generators, each corresponding to a back-off model, are implemented in other t\_\*.m modules such as t\_preserve\_all.m or t\_word\_for\_word\_factored.m. There, the abstract t\_generic interface is filled, and more importantly, the specific procedure of translation option generation is coded. The functionality is also used in fstatefstate.m, a module for probabilistic mapping of frontier states.

- bitree\_corpus.m implements routines for loading word-aligned parallel dependency treebank, including heuristics for constructing more coarse-grained Czech morphological tagsets from Bojar (2007).
- compact\_lt.m and treelet define the data type for a little tree and a treelet. A little tree keeps track of the structure of the tree, labels of internal and frontier nodes, and mutual order of sons of an internal node, including the position of the father among the sons. A treelet represents also the root label of the little tree.
- bilt.m defines the data type for a pair of dependency trees as loaded from a parallel treebank (the type t(LabelOfInternals,LabelOfFrontiers)) as well as the type for a pair of little trees extracted from the trees (the type bilt).
- linearizer.m implements line-oriented bi-pipe interface to external tree linearizers. The linearizer is responsible for converting a dependency tree to a plaintext representation of the sentence. We currently support two t-to-text linearizers: Ptáček and Žabokrtský (2006) and Žabokrtský (2008).
- lm.m defines the configuration options for *n*-gram language models and provides routines for scoring sequences of strings using IRST LM.
- binodelm.m defines the configuration options for bi-node language models, implements routines for collecting maximum-likelihood estimates of bi-node LM probabilities and provides routines for scoring trees using bi-node LMs.
- oc\_generic.m defines the type class for collecting decoder output (output collectors, hence oc). The decoder is currently capable of collecting the output either as dependency trees (oc\_tree.m) or directly linearized output (oc\_segments). This abstraction and the fact that language models are applied to decoder output only (ignoring the original tree) allows to use the appropriate language model for output collectors (i.e. lm.m for os\_segments.m and binodelm.m for oc\_tree.m).
- scorers.m provides an API for consistent handling of model weights (lambdas) and hypothesis scores. Each model used in an experiment has to register to obtain an identifier describing the lambdas the model needs. When the model assigns a score to a hypothesis, the identifier is used to find the appropriate element of the score vector.
- evaluation.m defines the configuration options for evaluating MT quality against some reference translation. The module currently supports evaluation using BLEU (for output collected as segments, oc\_segments.m, or for tree output linearized with an external linearizer), and using tree edit distance (for tree output, oc\_tree.m).
- bleu.m implements the BLEU metric of MT quality, including a pre-computation for efficient MERT optimization of weights on an *n*-best list of translations.

- drawing.m implements routines to display various data structures (e.g. a tree or a little tree) in SVG for the purposes of graphical representation of the log.
- mert.m is the Mercury interface to Philipp Koehn's C/C++ implementation of minimum errorrate training.

Some of TREEDECODE modules are rather generic and implement various useful data structures and algorithms:

- grid\_feeder.m is an abstract line-orientated file reader that may be run on SGE. To implement a specific reader, the user has to provide the main predicate used to observe data from one line of the file, as well as a few support predicates to set up a blank memory for data collection and to serialize and merge filled memories. Depending on the environment variable GRIDJOBS, the file is read in the current process, or a set of slave processes is submitted to the cluster, each collecting counts in a separate section of the file.
- tinycdb.m is a Mercury interface to tinycdb by Michael Tokarev, a very time efficient library for on-disk hashing. Both the key and the value can be arbitrary sequence of bytes. The database is "constant" (i.e. read-only) once created.
- hugemap.m implements an abstract data type for mapping keys to values. While each key-value pair has to fit to memory, the whole mapping is stored on disk and it does not need fit to memory. The map is constructed during a "growing" phase: as new keys and values are added to the map, either a RAM cache is updated, or they are saved to a journal. This can be combined with grid\_feeder.m to prepare several journals in parallel. At the end of the growing phase, all the journals are merged (multiple values attached to a single key are joined using a user-supplied predicate) and the resulting mapping is stored as a tinycdb. For the purpose of journalling and storing items in the tinycdb, the user has to provide also serialization predicates for both the key and the value.
- hmp\_map.m extends the functionality of hugemap to provide not only values for a given key, but also arbitrary user-defined probabilities estimated using MLE from some observed counts. The user has to specify: (1) what constitutes the key (the left-hand side of the mapping) and the value (the right-hand side of the mapping) given an "event", i.e. an observation; (2) what various counts should be collected from the corpus (e.g. the number of observations, the total number of observations of the key); and (3) final formulas to calculate the probabilities for all left-right pairs from the collected counts (e.g. the count of left-right pairs divided by the count of left observations).

A similar functionality was provided by the modules biprobdb.m and multiprob\_map.m. These are now obsolete and should be avoided in further development.

- bmp.m, bmp\_unlimited.m and wisebmp.m implement a bitmap. While bmp.m uses a single integer to store the bitmap and it is thus limited by the architecture to 32 or 64 bits, bmp\_unlimited.m uses sparse bitset from Mercury standard library. The module wisebmp.m combines the two by using the more compact representation as long as possible and resorting to the unlimited case once higher bits are needed.

initable.m provides a unified access to structures that are often needed in two forms: a serializable "configuration" and a non-serializable "inited" form. For example, a program pathname and some command line arguments is a configuration while a bi-directional pipe to the process launched using the arguments is the "inited" form that cannot be serialized. While the inited form can be unambiguously constructed from the configuration, the reverse is not possible in general. The module initable.m keeps track of both instatiations of the structure, allowing to implement serialization and deserialization of non-serializable things by serializing the configuration instead and re-using the inited form after deserialization.

From the user's perspective, if you have some objects that take the two forms: the serializable configuration (of type Config) and the inited version (of type Inited), use the type initable(Config, Inited) to refer to either of them. The module initable.m then provides an easy access to both of the variants as needed using the functions det\_inited/1 and det\_config/1. Before being able to access the inited form using det\_inited/1, be sure to call ensure\_inited/5 at least once.

Another advantage of using initable.m is that ensure\_inited/5 keeps and internal cache and avoids duplicit initialization if the Config was already inited. If you wish to indeed initialize some structure several times (such as opening several bi-directional pipes to several processes), you need to include some distinguishing information in the Config.

### 3.5.3 Wishlist of Future Improvements

We plan to modify or implement these features in future releases of TREEDECODE:

- Get rid of complex\_factor and extend t\_preserve\_all to factored output.
- Introduce corpus "header files" that specify factor names instead of FactorsConfig.
- Separate runtime settings (i.e. the *DirConf* sections from the configuration file and some of the environment variables such as SVGLOG) to a runtime configuration file.
- Improve combination of translation option generators in merge\_all:
  - if two generators create the same output, keep only one, but merge the scores,
  - if an option is generated by only one generator, the scores for other generators should get the lowest probability, not the highest.
- Extend the range of options when loading training corpus:
  - k-best select (select the parse that best matches word alignment),
  - parse projection (construct target-side parse by projecting syntax structure from the source side).
- Finish the implementation of delayed output factor generation,
- Improve debugging output for error analysis:
  - Why this part of the reference tree was not created by our system?
    - \* words not available in training data,
    - \* available in training data, but inextractable (conflict of word alignment),
    - \* lower score than the solution we found.
  - SVG output with more details (unused translation options, ...)
  - Save SVG output after every sentence.
  - Analysis of combinatorial explosion:

- \* How many translation options and hypotheses are pruned?
- $\ast\,$  Which translation option generators explode most often and how much?
- $\ast~$  Which input words/nodes cause the explosion most often?

## Appendix A

## References

- Ondřej Bojar and Martin Čmejrek. 2007. Mathematical Model of Tree Transformations. Project Euromatrix - Deliverable 3.2, ÚFAL, Charles University, December.
- Ondřej Bojar, Miroslav Janíček, Zdeněk Žabokrtský, Pavel Češka, and Peter Beňa. 2008. CzEng 0.7: Parallel Corpus with Community-Supplied Translations. In Proceedings of the Sixth International Language Resources and Evaluation (LREC'08), Marrakech, Morocco, May. ELRA.
- Ondřej Bojar. 2007. English-to-Czech Factored Machine Translation. In Proceedings of the Second Workshop on Statistical Machine Translation, pages 232–239, Prague, Czech Republic, June. Association for Computational Linguistics.
- Martin Čmejrek. 2006. Using Dependency Tree Structure for Czech-English Machine Translation. Ph.D. thesis, ÚFAL, MFF UK, Prague, Czech Republic.
- Marcello Federico and Mauro Cettolo. 2007. Efficient Handling of N-gram Language Models for Statistical Machine Translation. In *Proceedings of the Second Workshop on Statistical Machine Translation*, pages 88–95, Prague, Czech Republic, June. Association for Computational Linguistics.
- Marie Mikulová, Alevtina Bémová, Jan Hajič, Eva Hajičová, Jiří Havelka, Veronika Kolářová, Lucie Kučová, Markéta Lopatková, Petr Pajas, Jarmila Panevová, Magda Ševčíková, Petr Sgall, Jan Štěpánek, Zdeňka Urešová, Kateřina Veselá, and Zdeněk Žabokrtský. 2007. Annotation on the tectogrammatical level in the Prague Dependency Treebank. Project Euromatrix - Deliverable 3.1, ÚFAL, Charles University, May.
- Franz Josef Och. 2003. Minimum Error Rate Training in Statistical Machine Translation. In Proc. of the Association for Computational Linguistics, Sapporo, Japan, July 6-7.
- Jan Ptáček and Zdeněk Žabokrtský. 2006. Synthesis of Czech Sentences from Tectogrammatical Trees. In *Proc. of TSD*, pages 221–228.
- Dennis Shasha and Kaizhong Zhang. 1997. Approximate tree pattern matching. In *Pattern Matching Algorithms*, pages 341–371. Oxford University Press.
- Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1995. Mercury: An Efficient Purely Declarative Logic Programming Language. In Proceedings of the Australian Computer Science Conference, pages 499–512, Glenelg, Australia, February.
- Zdeněk Žabokrtský. 2008. Tecto MT. Technical report, ÚFAL/CKL, Prague, Czech Republic. In prep.

### Appendix B

## Example TreeDecode Configuration File

See Section 3.2.3 for reference.

```
\% Sample configuration file demonstrating basic tree-to-tree translation.
```

```
%%%%% Setting Variables:
```

```
% How many partial hypotheses are considered when constructing translation
% options step by step.
FactoredStackSize = int(10).
```

```
% Runtime options: where to store cached partial results, finished models
% and temporary files.
DirConf = dirconf(dirconf(
    cache_desc("$TREEDECODEROOT/cache", yes(1000)),
    "$TREEDECODEROOT/models",
    write_directly,
    fixed_scratch_space("/tmp")
  )).
```

```
% Interpretation of factors on the English (source) side of our corpus:
EnFactorsConfig = factors_config(factors_config(
 verbatim(3), \% the dependency structure is stored in factor 3
 wordfactor([
    "t_lemma"
                   - verbatim(0),
    "functor"
                  - verbatim(1),
    "gram/sempos" - verbatim(4),
    "gram/number" - verbatim(5),
    "gram/negation" - verbatim(6),
    "gram/tense"
                 - verbatim(7),
    "gram/verbmod" - verbatim(8),
    "gram/deontmod" - verbatim(9)
 ]),
 fstatefactor( % the frontier state is taken verbatim from factor 1
    verbatim(1)
 )
)).
% This is just a summary of all known English factors.
```

```
EnFactAll = factors_to_use(
       Γ
          "t_lemma",
          "functor",
          "gram/sempos",
          "gram/number",
          "gram/negation",
          "gram/tense",
          "gram/verbmod",
          "gram/deontmod"
        ]).
% A subset of English factors for back-off models.
EnFactLemFunc = factors_to_use(
        Γ
          "t_lemma",
          "functor"
        ]).
% Interpretation of Czech (i.e. target) side of our corpus:
CsFactorsConfig = factors_config(factors_config(
  verbatim(3), % the dependency structure is stored in factor 3
 wordfactor([
    % There is only one output factor that combines all attributes
   \% of Czech nodes.
    "all_in_one" - complex([
                     - 0,
      "t_lemma"
      "functor"
                      - 1,
      "gram/sempos" - 4,
      "gram/number"
                    - 5,
      "gram/negation" - 6,
                     - 7,
      "gram/tense"
      "gram/verbmod" - 8,
      "gram/deontmod" - 9,
      "val_frame.rf" - 10,
      "gram/indeftype" - 11,
      "subfunctor" - 12,
      "nodetype" - 13,
      "gram/aspect" - 14,
      "gram/numertype" - 15
   ])
 ]),
 fstatefactor( % the frontier state is taken verbatim from factor 1
    verbatim(1)
 )
)).
% This is just a summary of all known Czech factors:
CsFactAll = factors_to_use( [ "all_in_one" ] ).
% The training corpus file and description of its factors:
TrainCorpus = corpus(corpus(
          "$TREEDECODEROOT/corpora/basic_tecto_to_tecto.train.wali",
          EnFactorsConfig,
```

```
CsFactorsConfig
```

```
)).
% Configuration of a probabilistic mapping from source frontier states
% to target frontier states.
FFTable = fftable(not_inited(config(
  "ff", % The model will use weights labelled using the prefix 'ff'
 DirConf, % Use the common cache and model directories.
 TrainCorpus, % Use the common training corpus to train the model.
 3, % How many target fstates per source fstate should be considered.
 % Condition the mapping on the following factors:
  ["t_lemma"], % of the source node's parent
  ["all_in_one"], % of the target node's parent
 backoff_to_unconditioned, % but also back off to the unconditioned
                            % distribution if no detailed mapping is available.
 yes("default-fstate") % Ultimate back-off value for unseen source frontier
                        % states.
))).
% We allow at most 2 internal nodes, 4 frontier nodes and branching of 10
% in the demo.
Limits = limits(limits(2,4,10)).
%%%%% Main configuration term:
config(
 produce_tree, % Output is saved as a tree.
 Limits, % Use the Limits defined above when decomposing input trees.
  100, % Stacklimit, at most 100 hypotheses covering the same number of words
       % are expanded.
 % List of tweaks:
  Г
   tropt_limit - i(100) % Translation options limit.
 ],
  [], % Models weights, the empty list indicates to use the default setting.
 % Do not use any Minimum Error-Rate Training.
 no_mert,
 % Input file:
 file("$TREEDECODEROOT/corpora/basic_tecto_to_tecto.dev.en.in"),
 % The interpretation of factors in the input file:
 EnFactorsConfig,
 % Directly evaluate the output:
  evaluate_trees( % We're producing trees, so evaluate trees.
    treedist(config( % Evaluate using tree distance:
      % The reference file:
      "$TREEDECODEROOT/corpora/basic_tecto_to_tecto.dev.cs.ref",
```

```
% Interpretation of factors of the reference file:
CsFactorsConfig,
```

```
% Evaluate tree distance several times, each time using a different
   % subset of output factors.
   eval_factors([
     all_factors,
     section_of_complex_factor(0, "t_lemma"),
     section_of_complex_factor(0, "functor")
   ])
 )).
 % Do not linearize the output trees, do not evaluate the linearized form.
 no_linearized_evaluation
),
DirConf, % Use the common cache and temporary directory.
opened_positions_equal, % Two hypotheses are recombined if they share
 \% the set of pending frontier nodes (including their state labels).
% Translation options generators (models) to use:
first_of([ % Use the first model that produces an output.
 trg(
   % Translate a whole treelet at once, preserving the structure,
   % internal labels, frontier states and ordering of the nodes.
   t_preserve_all(config(
      "tr", % The model will use weights with label prefixed by "tr"
     DirConf, \% Use the common cache, model and temp directory
     TrainCorpus, % Use the common training corpus.
     EnFactAll, % Use all source factors.
     Limits, % Use the Limits defined above when training the model.
     climb_linked_roots_covered_internals, % A heuristic for finding
        % corresponding treelets in the training corpus.
     forbid_dangling_nodes, % Further refinement of the heuristic.
     ali_compatible, % Further refinement of the heuristic.
     keep_all_adjs, % Don't drop any frontier nodes, even for "adjuncts".
     no_conversion % Don't modify observed treelets in any way (e.g.
        % disregarding node order by sorting the treelets first).
   ))),
 trg(
   % Translate treelets consisting of single internal nodes:
   t_word_for_word_factored(config(
      "wfw", % The model will use weights with label prefixed by "wfw"
     DirConf, % Use the common cache, model and temp directory
     TrainCorpus, % Use the common training corpus.
     % Translation steps: there is only one step, translate all English
     % factors to all Czech factors.
      [ translate([], EnFactAll, CsFactAll) ],
     target_fstate_probabilistically, % The root state of the target
        % treelet is translated using the FFTable.
     FFTable, % The above defined probabilistic mapping of frontier states.
        \% It is used both for all sons of the translated internal node as
       % well as for the root state of the target treelet.
     FactoredStackSize % Use the common stack size when generating
       % output options.
   ))),
```

```
trg(
```

```
% Translate treelets consisting of single internal nodes:
      t_word_for_word_factored(config(
        "wfwlemfunc", % The prefix for weight labels.
        DirConf, % Use the common cache, model and temp directory
        TrainCorpus, % Use the common training corpus.
        % Translation steps: use English lemma to guess all Czech factors.
        [ translate([], EnFactLemFunc, CsFactAll) ],
        target_fstate_probabilistically, % see above
        FFTable, % see above
        FactoredStackSize % see above
     ))),
    trg(
     % Back-off: keep the lemma of an unknown node untranslated:
     % The back-off model is actually implemented using the
     % t_word_for_word_factored
      t_word_for_word_factored(config(
        "unk", % The prefix for weight labels.
        DirConf, % see above
        TrainCorpus, % see above
        % Translation steps: just copy English t_lemma to the single Czech
        % (output) factor/
        [ copy(["t_lemma"], ["all_in_one"]) ],
        target_fstate_probabilistically, % see above
        FFTable, % see above
        FactoredStackSize % see above
      )))
 ]),
 % Details on output:
  oc_tree(config(
    just_factor(0), % When producing final output tree, which factors should be
      % exported?
    % The list of language models to use:
    Γ
       % This is a sample binode LM configuration.
       % Should you face memory problems, just comment this section out,
       % the model is can be too big because it is based on all Czech atributes
       % at once and not restricted to e.g. the t_lemma only.
     not_inited(config(
        "binodelm", % The prefix for weight labels.
        DirConf, % see above
        TrainCorpus, % see above
        [ "all_in_one" ], % Attributes of the dependent node to consider.
        [ "all_in_one" ] % Attributes of the governing node to consider.
     ))
   ],
   no_linearized_rescoring % Do not linearize output trees, do not rescore
                            % them using any n-gram language model.
 ))
).
```