MOSES



Statistical Machine Translation System

User Manual and Code Guide

Philipp Koehn
pkoehn@inf.ed.ac.uk
University of Edinburgh

Abstract

This document serves as user manual and code guide for the Moses machine translation decoder. The decoder was mainly developed by Hieu Hoang and Philipp Koehn at the University of Edinburgh and extended during a Johns Hopkins University Summer Workshop and further developed under EuroMatrix and GALE project funding. The decoder (which is part of a complete statistical machine translation toolkit) is the de facto benchmark for research in the field.

This document serves two purposes: a user manual for the functions of the Moses decoder and a code guide for developers. In large parts, this manual is identical to documentation available at the official Moses decoder web site http://www.statmt.org/. This document does not describe in depth the underlying methods, which are described in the text book *Statistical Machine Translation* (Philipp Koehn, Cambridge University Press, 2009).

February 28, 2010

Acknowledgments

The Moses decoder was supported by the European Framework 6 projects EuroMatrix, TC-Star, the European Framework 7 project EuroMatrixPlus, and the DARPA GALE project, as well as several universities such as the University of Edinburgh, the University of Maryland, ITC-irst, Massachusetts Institute of Technology, and others.

Contributors are too many to mention, but it is important to stress the substantial contributions from Hieu Hoang, Chris Dyer, Josh Schroeder, Marcello Federico, Richard Zens, and Wade Shen. Moses is an open source project under the guidance of Philipp Koehn.

Contents

1 Introduction **9**

1.1 Welcome to Moses! . . . 9

1.1.1 Features . . . 9

1.1.2 Get started . . . 9

1.1.3 Acknowledgement . . . 10

1.1.4 Open Source License . . . 10

1.2 Getting Started with the Moses Software . . . 10

1.2.1 Check out the source code . . . 10

1.2.2 Note on code branches . . . 11

1.2.3 Required additional software . . . 11

1.2.4 Compile . . . 11

1.2.5 Run it for the first time . . . 12

1.2.6 Compiling Chart Decoder . . . 12

1.2.7 Ubuntu NLP Repository . . . 13

1.3 Roadmap . . . 13

2 Decoding Manual **15**

2.1 Tutorial . . . 15

2.1.1 Content . . . 15

2.1.2 A Simple Translation Model . . . 15

2.1.3 Running the Decoder . . . 16

2.1.4 Trace . . . 18

2.1.5 Verbose . . . 19

2.1.6 Tuning for Quality . . . 22

2.1.7 Tuning for Speed . . . 23

2.1.8 Limit on Distortion (Reordering) . . . 26

2.2 Tutorial for Using Factored Models . . . 27

2.2.1 Train a unfactored model . . . 28

2.2.2 Train a model with POS tags . . . 28

2.2.3 Train a model with generation and translation steps . . . 31

2.2.4 Train a morphological analysis and generation model . . . 32

2.2.5 Train a model with multiple decoding paths . . . 33

2.3 Syntax Tutorial . . . 34
2.3.1 Tree-Based Models . . . 35
2.3.2 Decoding . . . 37
2.3.3 Decoder Parameters . . . 40
2.3.4 Training . . . 41
2.4 Support Tools . . . 46
2.4.1 Overview . . . 46
2.4.2 Content . . . 46
2.4.3 Converting Pharaoh configuration files to Moses configuration files . . . 46
2.4.4 Moses decoder in parallel . . . 47
2.4.5 Filtering phrase tables for Moses . . . 47
2.4.6 Reducing and Extending the Number of Factors . . . 48
2.4.7 Scoring translations with BLEU . . . 48
2.4.8 Missing and Extra N-Grams . . . 48
2.4.9 Making a Full Local Clone of Moses Model + ini File . . . 48
2.4.10 Absolutizing Paths in moses.ini . . . 49
2.4.11 Printing Statistics about Model Components . . . 49
2.4.12 Recaser . . . 50
2.4.13 Truecaser . . . 50
2.4.14 Searchgraph to DOT . . . 51
2.5 Advanced Features of the Decoder . . . 51
2.5.1 Content . . . 51
2.5.2 Lexicalized Reordering Models . . . 52
2.5.3 Binary Phrase Tables with On-demand Loading . . . 54
2.5.4 Binary Reordering Tables with On-demand Loading . . . 55
2.5.5 XML Markup . . . 55
2.5.6 Generating n-Best Lists . . . 57
2.5.7 Word-to-word alignment . . . 58
2.5.8 Minimum Bayes Risk Decoding . . . 59
2.5.9 Handling Unknown Words . . . 60
2.5.10 Output Search Graph . . . 60
2.5.11 Early Discarding of Hypotheses . . . 62
2.5.12 Maintaining stack diversity . . . 63
2.5.13 Cube Pruning . . . 63
2.5.14 Specifying Reordering Constraints . . . 64
2.5.15 Multiple Translation Tables . . . 65
2.5.16 Pruning the Translation Table . . . 66
2.5.17 Multi-threaded Moses . . . 67
2.5.18 Moses Server . . . 68
2.5.19 Amazon EC2 cloud . . . 68

2.6 Translating Web pages with Moses . . . 68
2.6.1 Introduction . . . 69
2.6.2 Detailed setup instructions . . . 71

3 Training Manual **75**
3.1 Training . . . 75
3.1.1 Content . . . 75
3.1.2 Training process . . . 75
3.1.3 Running the training script . . . 76
3.2 Preparing Training Data . . . 76
3.2.1 Training data for factored models . . . 77
3.2.2 Cleaning the corpus . . . 77
3.3 Factored Training . . . 78
3.3.1 Translation factors . . . 78
3.3.2 Reordering factors . . . 79
3.3.3 Generation factors . . . 79
3.3.4 Decoding steps . . . 79
3.4 Training Step 1: Prepare Data . . . 79
3.5 Training Step 2: Run GIZA++ . . . 81
3.5.1 Training on really large corpora . . . 82
3.5.2 Training in parallel . . . 82
3.6 Training Step 3: Align Words . . . 82
3.7 Training Step 4: Get Lexical Translation Table . . . 85
3.8 Training Step 5: Extract Phrases . . . 85
3.9 Training Step 6: Score Phrases . . . 85
3.10 Training Step 7: Build reordering model . . . 87
3.11 Training Step 8: Build generation model . . . 88
3.12 Training Step 9: Create Configuration File . . . 88
3.13 Building a Language Model . . . 89
3.13.1 Content . . . 89
3.13.2 Language Models in Moses . . . 89
3.13.3 Building a LM with the SRI LM Toolkit . . . 90
3.13.4 On the IRST LM Toolkit . . . 90
3.13.5 RandLM . . . 94
3.14 Tuning . . . 98
3.14.1 More on the lambda settings: . . . 101
3.14.2 Tuning on a subset of features . . . 101
3.14.3 Tuning on a subset of features (Old version) . . . 102
3.14.4 Running the script on a cluster . . . 104

4 Background **105**

4.1 Background . . . 105
4.1.1 Model . . . 106
4.1.2 Word Alignment . . . 106
4.1.3 Methods for Learning Phrase Translations . . . 107
4.1.4 Och and Ney . . . 108
4.2 Decoder . . . 110
4.2.1 Translation Options . . . 111
4.2.2 Core Algorithm . . . 111
4.2.3 Recombining Hypotheses . . . 113
4.2.4 Beam Search . . . 113
4.2.5 Future Cost Estimation . . . 115
4.2.6 N-Best Lists Generation . . . 116
4.3 Factored Translation Models . . . 118
4.3.1 Motivating Example: Morphology . . . 119
4.3.2 Decomposition of Factored Translation . . . 120
4.3.3 Statistical Model . . . 121
4.4 Confusion Networks Decoding . . . 124
4.4.1 Confusion Networks . . . 124
4.4.2 Representation of Confusion Network . . . 125
4.5 Word Lattices . . . 126
4.5.1 How to represent lattice inputs . . . 126
4.5.2 Configuring moses to translate lattices . . . 127
4.6 Publications . . . 127

5 Code Guide **129**
5.1 Code Guide . . . 129
5.1.1 Quick Start . . . 129
5.2 Coding Style . . . 130
5.3 Factors, Words, Phrases . . . 131
5.3.1 Factors . . . 131
5.3.2 Words . . . 132
5.3.3 Factor Types . . . 132
5.3.4 Phrases . . . 132
5.4 Adding Feature Functions . . . 133
5.4.1 Score Producer . . . 134
5.4.2 Adding Decoder Parameters . . . 136
5.4.3 Integrating the Feature . . . 136
5.4.4 Implementing the Feature . . . 137
5.4.5 Tuning . . . 138
5.5 Regression Testing . . . 139

5.5.1 Goals . . . 139
5.5.2 Test suite . . . 139
5.5.3 Running the test suite . . . 140
5.5.4 How it works . . . 140
5.5.5 Writing regression tests . . . 141

6 Reference **143**
6.1 Frequently Asked Questions . . . 143
6.1.1 Content . . . 143
6.1.2 My system is taking a really long time to translate a sentence. What can I do to speed it up ? . . . 144
6.1.3 The system runs out of memory during decoding. . . . 144
6.1.4 I would like to point out a bug / contribute code. . . . 144
6.1.5 How can I get an updated version of Moses ? . . . 144
6.1.6 When is version 1/beta of Moses coming out? . . . 145
6.1.7 I'm an undergrad/masters student looking for a project in SMT. What should I do? . . . 145
6.1.8 What do the 5 numbers in the phrase table mean? . . . 145
6.1.9 What OS does Moses run on? . . . 145
6.1.10 Can I use Moses on Windows ? . . . 146
6.1.11 Do I need a computer cluster to run experiments? . . . 146
6.1.12 I've compiled Moses, but it segfaults when running. . . . 146
6.1.13 How do I add a new feature function to the decoder? . . . 146
6.1.14 Compiling with SRILM or IRSTLM produces errors. . . . 146
6.1.15 I'm trying to use Moses to create a web page to do translation. . . . 147
6.1.16 How can a create a system that translate both ways, ie. X-to-Y as well as Y-to-X ? . . . 147
6.1.17 PhraseScore dies with signal 11 - why? . . . 147
6.1.18 Does Moses do Hierarchical decoding, like Hiero etc? . . . 148
6.1.19 Can I use Moses in proprietary software ? . . . 148
6.1.20 Who do I ask if my question hasn't been answered by this FAQ? . . . 148
6.2 Reference: All Decoder Parameters . . . 148
6.3 Reference: All Training Parameters . . . 151
6.3.1 Basic Options . . . 152
6.3.2 Factored Translation Model Settings . . . 154
6.3.3 Lexicalized Reordering Model . . . 154
6.3.4 Partial Training . . . 155
6.3.5 File Locations . . . 155
6.3.6 Alignment Heuristic . . . 156
6.3.7 Maximum Phrase Length . . . 157

6.3.8 GIZA++ Options 157
6.3.9 Dealing with large training corpora 157

1

Introduction

1.1 Welcome to Moses!

Moses is a **statistical machine translation system** that allows you to automatically train translation models for any language pair. All you need is a collection of translated texts (parallel corpus). An efficient search algorithm finds quickly the highest probability translation among the exponential number of choices.

1.1.1 Features

- Moses allows the decoding of **confusion networks (page 124)** and **word lattices (page 125)**, enabling easy integration with ambiguous upstream tools, such as automatic speech recognizers or morphological analyzers
- Moses offers two types of translation models: **phrase-based (page 15)** and **tree-based (page 34)**
- Moses features **factored translation models (page 27)**, which enable the integration linguistic and other information at the word level

1.1.2 Get started

The released software includes a command line executable which can used for decoding. The source code for the decoder, as well as binaries for Windows and Linux, can be downloaded from Sourceforge. Download the latest binary release[1]. However, if you want the addition scripts to create factored phrase tables and train the weights of the models, please download the complete snapshot via SVN[2]. This repository also contains regression tests, should you be interested in enhancing the decoder.

[1] http://mosesdecoder.sourceforge.net/download.php
[2] http://mosesdecoder.sourceforge.net/svn.php

Learn about the decoder (page 15), training models (page 75), and tuning (page 98). Follow the step-by-step guide[3]. The documentation available at this web side is also compiled in a printable manual[4].

1.1.3 Acknowledgement

The development of Moses is mainly supported under the EuroMatrix[5] and EuroMatrixPlus[6] projects, funded by the European Commission under Framework Programme 6 and 7, and received additional support from

- University of Edinburgh, Scotland
- RWTH Aachen, Germany
- Fondazione Bruno Kessler, Trento, Italy
- University of Maryland, College Park, United States
- Massachusetts Institute of Technology, United States
- Charles University, Prague, Czech Republic
- US funding agencies DARPA, NSF, and Department of Defence
- EU funding through the TC-Star project

1.1.4 Open Source License

Moses is licensed under the LGPL[7].

1.2 Getting Started with the Moses Software

1.2.1 Check out the source code

The source code is stored in a Subversion repository on Sourceforge[8].

You should check out the two main modules with the following commands:

```
mkdir ~/mosesdecoder
cd ~
svn cos://svn.sourceforge.net/svnroot/mosesdecoder/trunk mosesdecoder
```

[3] http://www.statmt.org/moses_steps.html
[4] http://www.statmt.org/moses/manual/manual.pdf
[5] http://www.euromatrix.net/
[6] http://www.euromatrixplus.net/
[7] http://www.gnu.org/licenses/lgpl.html
[8] http://sourceforge.net/svn/?group_id=171520

1.2.2 Note on code branches

Subversion handles branches differently than CVS (more along the lines of systems like Perforce), in that they are just directories in the main repository for a project. By convention, the main branch is stored in `svnroot/trunk` and branches are under `svnroot/branches/*`. The command above checks out the main line only.

1.2.3 Required additional software

Moses requires a language model toolkit to build. At the moment, Moses supports:

- the SRILM language modeling toolkit[9]
- the IRST language modeling toolkit[10]

These toolkits provide commands to estimate and compile language models, and are in some way complementary. See the corresponding documentation to learn more about their main features.

Moses provides interfaces towards them. According to your need, you can choose one of them, but we suggest to get both.

Download and install them in your favourite location.

1.2.4 Compile

After you check out from Subversion, you'll need to run the following script in the `mosesdecoder` directory.

```
cd ~/mosesdecoder
./regenerate-makefiles.sh
```

Note: Versions 1.9 (or higher) of aclocal and automake are required. Note: For Mac OSX users:

Standard distribution usually includes versions 1.6. Get versions 1.9 (or higher) and set the variables ACLOCAL and AUTOMAKE in ./regenerate-makefiles.sh.

Next, configure and build the moses executable. At this point, you should choose the preferred LM toolkit. To do that use the parameter either `-with-srilm` or `-with-irstlm` as follows.

```
cd ~/mosesdecoder
./configure --with-srilm=/path-to-srilm (or --with-irstlm=/path-to-irstlm)
make
```

[9] http://www.speech.sri.com/projects/srilm/
[10] http://sourceforge.net/projects/irstlm/

NB. You CAN set both options to build moses with support for both IRST and SRI LM.

When you add a file to the project, you need to add it to `$PROJ/Makefile.am` and run autoconf and automake from `$PROJ` before running `./configure` again. (`$PROJ` is `/mosesdecoder/moses/` or `/mosesdecoder/moses-cmd/`.)

1.2.5 Run it for the first time

Download the sample models and extract them into your working directory:

```
cd ~/mosesdecoder
wget://www.statmt.org/moses/download/sample-models.tgz
tar xzf sample-models.tgz
cd sameple-models/phrase-model

~/mosesdecoder/moses-cmd/src/moses -f moses.ini < in > out
```

If everything worked out right, this should translate the sentence `das ist ein kleines haus` (in the file `in`) as `it is a small house` (in the file `out`).

1.2.6 Compiling Chart Decoder

The code is currently not part of the main trunk distribution, but in a special branch in the Sourceforge SVN distribution.

To obtain the code, you will need to check it out from Sourceforge:

```
svn cos://mosesdecoder.svn.sourceforge.net/svnroot/mosesdecoder/branches/mt3_chart moses-ch
```

To compile

```
./regenerate-makefiles.sh
./configure --with-srilm=... --with-irstlm=... --with-berkeleydb=/usr/local/BerkeleyDB.4.7
```

XCode and Visual Studio project files also works.

Berkeley db is used for the new binary phrase table. The flag is NOT optional and should point to the root where the include files AND the lib files are stored.

You can get berkeley db from the internet. When compiling it, make sure it is compiled to support C++ too. The files `libdb_cxx-4.7...` should be created along with `libdb-4.7`.

Email Hieu Hoang[11] if you have any questions.

The compiled decoder binary is in `moses-chart-cmd/src/moses_chart`.

Internal Comment

For Edinburgh peps, the 64-bit Berkeley db libs can be found on Eddie:

[11] http://groups.inf.ed.ac.uk/hoang/hieu/

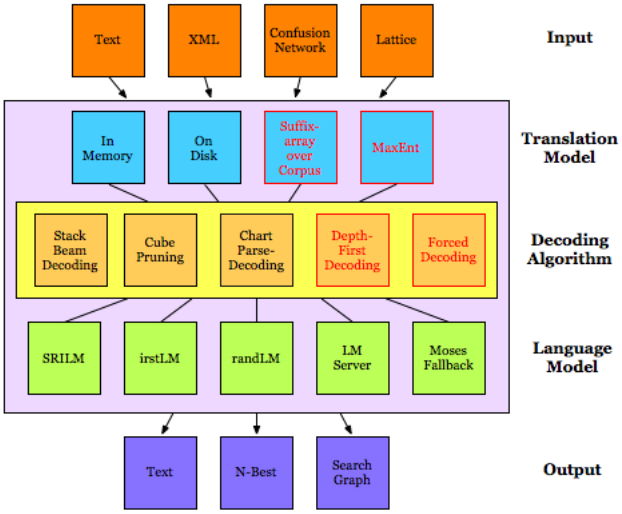
```
/exports/informatics/inf_iccs_smt/shared/BerkeleyDb
```

1.2.7 Ubuntu NLP Repository

Alternatively, Eric Nichols has put together an Ubuntu package which makes it easier to install moses. It can be downloaded from http://cl.naist.jp/ eric-n/ubuntu-nlp/.

1.3 Roadmap

The Moses decoder follows a very modular design, and we are continuously extending its capabilities. See below for a diagram that illustrates the different input and output modalities, as well as different decoding algorithms supported by different implementations of language models and translation models.



The boxes with red text are currently planned extension:

- **Depth-first decoding**: to provide anytime algorithms for decoding
- **Forced decoding**: to compute scores for provided output
- **Suffix-array translation models**: an alternative way to store large rule-sets without the need to translate them
- **Maximum entropy translation models**: translation models that incorporate additional source-side and context information for scoring translation rules.

Since most of this work is carried out in academic settings, it is hard to plan when it will be completed, but we expect that most of the capabilities mentioned above will be integrated in 2009.

Notable additions in 2009

- Tree-based models
- Multi-threaded decoding
- Language model server

Notable additions in 2008

- Specification of reordering constraints with XML
- Early discarding pruning
- Randomized language models
- Output of search graph
- Cube pruning

2

Decoding Manual

2.1 Tutorial

This tutorial describes the use of statistical machine translation decoder Moses. It assumes that you already have a trained model. For more information on training models, please refer to the training overview (page 75).

2.1.1 Content

- A Simple Translation Model (page 15)
- Running the Decoder (page 16)
- Trace (page 18)
- Verbose (page 19)
- Tuning for Quality (page 22)
- Tuning for Speed (page 23)
- Translation Table Size (page 23)
- Hypothesis Stack Size (Beam) (page 24)
- Limit on Distortion (Reordering) (page 26)

2.1.2 A Simple Translation Model

Let us begin with a look at the toy phrase-based translation model that is available for download at http://www.statmt.org/moses/download/sample-models.tgz. Unpack the tar ball and enter the directory `sample-models/phrase-model`.

The model consists of two files:

- `phrase-table` the phrase translation table
- `moses.ini` the configuration file for the decoder

Let us look at the first line of the **phrase translation table** (file `phrase-table`):

```
der ||| the ||| 0.3
```

This entry means that the probality of translating the English word `the` from the German `der` is 0.3. Or in mathematical notation: p(the|der)=0.3. Note that these translation probabilities are in the inverse order due to the noisy channel model.

The translation tables are the main knowledge source for the machine translation decoder. The decoder consults these tables to figure out how to translate input in one language into output in another language.

Being a phrase translation model, the translation tables do not only contain single word entries, but multi-word entries. These are called **phrases**, but this concept means nothing more than an arbitrary sequence of words, with no sophisticated linguistic motivation.

Here is an example for a phrase translation entry in `phrase-table`:

```
das ist ||| this is ||| 0.8
```

2.1.3 Running the Decoder

Without further ado, let us run the decoder:

```
% echo 'das ist ein kleines haus' | moses -f moses.ini > out
Moses (built on Aug 29 2006)
a beam search decoder for phrase-based statistical machine translation models
written by Hieu Hoang, with contributions by Nicola Bertoldi, Ondrej Bojar,
Chris Callison-Burch, Alexandra Constantin, Brooke Cowan, Chris Dyer, Marcello
Federico, Evan Herbst, Philipp Koehn, Christine Moran, Wade Shen, Richard Zens.
(c) 2006 University of Edinburgh, Scotland
command: ../../moses-cmd/src/moses -f moses.ini
Defined parameters (per moses.ini or switch):
       config: moses.ini
       input-factors: 0
       lmodel-file: 0 0 3 ../lm/europarl.srilm.gz
       mapping: T 0
       ttable-file: 0 0 1 phrase-table
       ttable-limit: 10
       weight-d: 1
       weight-l: 1
       weight-t: 1
       weight-w: 0
Start loading LanguageModel ../lm/europarl.srilm.gz : [0.00] seconds
Finished loading LanguageModels : [3.00] seconds
IO from STDOUT/STDIN
Start loading PhraseTable phrase-table : [3.00] seconds
Finished loading phrase tables : [3.00] seconds
```

```
Created input-output object : [3.00] seconds
End. : [3.00] seconds

% cat out
this is a small house
```

Here, the toy model managed to translate the German input sentence `das ist ein kleines haus` into the English `this is a small house`, which is a correct translation.

The decoder is controlled by the configuration file `moses.ini`. The file used in the example above is displayed below.

```
########################
### MOSES CONFIG FILE ###
########################

# input factors
[input-factors]
0

# mapping steps, either (T) translation or (G) generation
[mapping]
T 0

# translation tables: source-factors, target-factors, number of scores, file
[ttable-file]
0 0 1 phrase-table

# language models: type(srilm/irstlm), factors, order, file
[lmodel-file]
0 0 3 ../lm/europarl.srilm.gz

# limit on how many phrase translations e for each phrase f are loaded
[ttable-limit]
10

# distortion (reordering) weight
[weight-d]
1

# language model weights
[weight-l]
1

# translation model weights
[weight-t]
1
```

```
# word penalty
[weight-w]
0
```

We will take a look at all the parameters that are specified here (and then some) later. At this point, let us just note that the translation model files and the language model file are specified here. In this example, the file names are relative paths, but usually having full paths is better, so that the decoder does not have to be run from a specific directory.

All parameters can be specfied in this file, or on the command line. For instance, we can also indicate the language model file on the command line:

```
% moses -f moses.ini -lmodel-file "0 0 3 ../lm/europarl.srilm.gz"
```

We just ran the decoder on a single sentence provided on the command line. Usually we want to translate more than one sentence. In this case, the input sentences are stored in a file, one sentence per line. This file is piped into the decoder and the output is piped into some output file for further processing:

```
% moses -f moses.ini < in > out
```

2.1.4 Trace

How the decoder works is described in detail in the background (page 105) section. But let us first develop an intuition by looking under the hood. There are two switches that force the decoder to reveal more about its inner workings: `-report-segmentation` and `-verbose`.

The trace option reveals which phrase translations were used in the best translation found by the decoder. Running the decoder with the segmentation trace switch (short `-t`) on the same example

```
echo 'das ist ein kleines haus' | moses -f moses.ini -t >out
```

gives us the extended output

```
% cat out
this is |0-1| a |2-2| small |3-3| house |4-4|
```

Each generated English phrase is now annotated with additional information:

- `this is` was generated from the German words 0-1, `das ist`
- `a` was generated from the German words 2-2, `ein`,
- `small` was generated from the German words 3-3, `kleines`

- house was generated from the German words 4-4, haus

Note that the German sentence does not have to be translated in sequence. Here an example, were the English output is reordered:

```
echo 'ein haus ist das' | moses -f moses.ini -t -d 0
```

The output of this command is:

```
this |3-3| is |2-2| a |0-0| house |1-1|
```

2.1.5 Verbose

Now for the next switch, -verbose (short -v), that displays additional run time information. The verbosity of the decoder output exists in three levels. The default is 1. Moving on to -v 2 gives additional statistics for each translated sentences:

```
% echo 'das ist ein kleines haus' | moses -f moses.ini -v 2
[...]
TRANSLATING(1): das ist ein kleines haus
       Total translation options: 12
Total translation options pruned: 0
```

A short summary on how many translations options were used for the translation of these sentences.

```
Stack sizes: 1, 10, 2, 0, 0, 0
Stack sizes: 1, 10, 27, 6, 0, 0
Stack sizes: 1, 10, 27, 47, 6, 0
Stack sizes: 1, 10, 27, 47, 24, 1
Stack sizes: 1, 10, 27, 47, 24, 3
Stack sizes: 1, 10, 27, 47, 24, 3
```

The stack sizes after each iteration of the stack decoder. An iteration is the processing of all hypotheses on one stack: After the first iteration (processing the initial empty hypothesis), 10 hypothesis that cover one German word are placed on stack 1, and 2 hypotheses that cover two foreign words are placed on stack 2. Note how this relates to the 12 translation options.

```
total hypotheses generated = 453
         number recombined = 69
             number pruned = 0
    number discarded early = 272
```

During the beam search a large number of hypotheses are generated (453). Many are discarded early because they are deemed to be too bad (272), or pruned at some later stage (0), and some are recombined (69). The remainder survives on the stacks.

```
total source words = 5
     words deleted = 0 ()
    words inserted = 0 ()
```

Some additional information on word deletion and insertion, two advanced options that are not activated by default.

```
BEST TRANSLATION: this is a small house [11111]  [total=-28.923] <<0.000, -5.000, 0.000, -2
Sentence Decoding Time: : [4.000] seconds
```

And finally, the translated sentence, its coverage vector (all 5 bits for the 5 German input words are set), it overall log-probability score, and the breakdown of the score into language model, reordering model, word penalty and translation model components.

Also, the sentence decoding time is given.

The most verbose output `-v 3` provides even more information. In fact, it is so much, that we could not possibly fit it in this tutorial. Run the following command and enjoy:

```
% echo 'das ist ein kleines haus' | moses -f moses.ini -v 3
```

Let us look together at some highlights. The overall translation score is made up from several components. The decoder reports these components, in our case:

```
The score component vector looks like this:
   0 distortion score
   1 word penalty
   2 unknown word penalty
   3 3-gram LM score, factor-type=0, file=../lm/europarl.srilm.gz
   4 Translation score, file=phrase-table
```

Before decoding, the phrase translation table is consulted for possible phrase translations. For some phrases, we find entries, for others we find nothing. Here an excerpt:

```
[das ; 0-0]
       the , pC=-0.916, c=-5.789
       this , pC=-2.303, c=-8.002
       it , pC=-2.303, c=-8.076

[das ist ; 0-1]
       it is , pC=-1.609, c=-10.207
       this is , pC=-0.223, c=-10.291

[ist ; 1-1]
       is , pC=0.000, c=-4.922
       's , pC=0.000, c=-6.116
```

The pair of numbers next to a phrase is the coverage, `pC` denotes the log of the phrase translation probability, after `c` the future cost estimate for the phrase is given.

Future cost is an estimate of how hard it is to translate different parts of the sentence. After looking up phrase translation probabilities, future costs are computed for all contigous spans over the sentence:

```
future cost from 0 to 0 is -5.789
future cost from 0 to 1 is -10.207
future cost from 0 to 2 is -15.722
future cost from 0 to 3 is -25.443
future cost from 0 to 4 is -34.709
future cost from 1 to 1 is -4.922
future cost from 1 to 2 is -10.437
future cost from 1 to 3 is -20.158
future cost from 1 to 4 is -29.425
future cost from 2 to 2 is -5.515
future cost from 2 to 3 is -15.236
future cost from 2 to 4 is -24.502
future cost from 3 to 3 is -9.721
future cost from 3 to 4 is -18.987
future cost from 4 to 4 is -9.266
```

Some parts of the sentence are easier to translate than others. For instance the estimate for translating the first two words (0-1: `das ist`) is deemed to be cheaper (-10.207) than the last two (3-4: `kleines haus`, -18.987). Again, the negative numbers are log probabilities.

After all this preperation, we start to create partial translations by translating a phrase at a time. The first hypothesis is generated by translating the first German word as `the`:

```
creating hypothesis 1 from 0 ( <s> )
        base score 0.000
        covering 0-0: das
        translated as: the
        score -2.951 + future cost -29.425 = -32.375
        unweighted feature scores: <<0.000, -1.000, 0.000, -2.034, -0.916>>
added hyp to stack, best on stack, now size 1
```

Here, starting with the empty initial hypothesis 0, a new hypothesis (`id=1`) is created. Starting from zero cost (base score), translating the phrase `das` into `the` carries translation cost (-0.916), distortion or reordering cost (0), language model cost (-2.034), and word penalty (-1). Recall that the score component information is printed out earlier, so we are able to interpret the vector.

Overall, a weighted log-probility cost of -2.951 is accumulated. Together with the future cost estimate for the remaining part if the sentence (-29.425), this hypothesis is assigned a score of -32.375.

And so it continues, for a total of 453 created hypothesis. At the end, the best scoring final hypothesis is found and the hypothesis graph traversed backwards to retrieve the best translation:

```
Best path: 417 <= 285 <= 163 <= 5 <= 0
```

Confused enough yet? Before we get caught too much in the intricate details of the inner workings of the decoder, let us return to actually using it. Much of what has just been said will become much clearer after reading the background (page 105) information.

2.1.6 Tuning for Quality

The key to good translation performance is having a good phrase translation table. But some tuning can be done with the decoder. The most important is the tuning of the model parameters.

The probability cost that is assigned to a translation is a product of probability costs of four models:

- phrase translation table,
- language model,
- reordering model, and
- word penalty.

Each of these models contributes information over one aspect of the characteristics of a good translation:

- The **phrase translation** table ensures that the English phrases and the German phrases are good translations of each other.
- The **language model** ensures that the output is fluent English.
- The **distortion model** allows for reordering of the input sentence, but at a cost: The more reordering, the more expensive is the translation.
- The **word penalty** provides means to ensure that the translations do not get too long or too short.

Each of the components can be given a weight that sets its importance. Mathematically, the cost of translation is:

$$p(e|f) = \phi(f|e)^{\text{weight}_\phi} \times \text{LM}^{\text{weight}_{\text{LM}}} \times D(e,f)^{\text{weight}_d} \times W(e)^{\text{weight}_\phi} \tag{2.1}$$

The propability p(e|f) of the English translation e given the foreign input f is broken up into four models, phrase translation phi(f|e), language model LM(e), distortion model D(e,f), and word penalty W(e) = exp(length(e)). Each of the four model is weighted by a weight.

The weighting is provided to the decoder with the four parameters `weight-t`, `weight-l`, `weight-d`, and `weight-w`. The default setting for these weights are 1, 1, 1, and 0. These are also the values in the configuration file `moses.ini`.

Setting these weights to the right values can improve translation quality. We already sneaked in one example above. When translating the German sentence `ein haus ist das`, we set the distortion weight to 0 to get the right translation:

```
% echo 'ein haus ist das' | moses -f moses.ini -d 0
this is a house
```

With the default weights, the translation comes out wrong:

```
% echo 'ein haus ist das' | moses -f moses.ini
a house is the
```

What is the right weight setting depends on the corpus and the language pair. Ususally, a held out development set is used to optimize the parameter settings. The simplest method here is to try out with a large number of possible settings, and pick what works best. Good values for the weights for phrase translation table (`weight-t`, short `tm`), language model (`weight-l`, short `lm`), and reordering model (`weight-d`, short `d`) are 0.1-1, good values for the word penalty (`weight-w`, short `w`) are -3-3. Negative values for the word penalty favor longer output, positive values favor shorter output.

2.1.7 Tuning for Speed

Let us now look at some additional parameters that help to speed up the decoder. Unfortunately higher speed usually comes at cost of translation quality. The speed-ups are achieved by limiting the search space of the decoder. By cutting out part of the search space, we may not be able to find the best translation anymore.

Translation Table Size

One strategy to limit the search space is by reducing the number of translation options used for each input phrase, i.e. the number of phrase translation table entries that are retrieved. While in the toy example, the translation tables are very small, these can have thousands of entries per phrase in a realistic scenario. If the phrase translation table is learned from real data, it contains a lot of noise. So, we are really interested only in the most probable ones and would like to elimiate the others.

The are two ways to limit the translation table size: by a fixed limit on how many translation options are retrieved for each input phrase, and by a probability threshold, that specifies that the phrase translation probability has to be above some value.

Compare the statistics and the translation output for our toy model, when no translation table limit is used

```
% echo 'das ist ein kleines haus' | moses -f moses.ini -ttable-limit 0 -v 2
[...]
        Total translation options: 12
[...]
total hypotheses generated = 453
         number recombined = 69
             number pruned = 0
    number discarded early = 272
[...]
BEST TRANSLATION: this is a small house [11111]  [total=-28.923]
```

with the statistics and translation output, when a limit of 1 is used

```
% echo 'das ist ein kleines haus' | moses -f moses.ini -ttable-limit 1 -v 2
[...]
        Total translation options: 6
[...]
total hypotheses generated = 127
         number recombined = 8
             number pruned = 0
    number discarded early = 61
[...]
BEST TRANSLATION: it is a small house [11111]  [total=-30.327]
```

Reducing the number of translation options to only one per phrase, had a number of effects: (1) Overall only 6 translation options instead of 12 translation options were collected. (2) The number of generated hypothesis fell to 127 from 442, and no hypotheses were pruned out. (3) The translation changed, and the output now has lower log-probability: -30.327 vs. -28.923.

Hypothesis Stack Size (Beam)

A different way to reduce the search is to reduce the size of hypothesis stacks. For each number of foreign words translated, the decoder keeps a stack of the best (partial) translations. By reducing this stack size the search will be quicker, since less hypotheses are kept at each stage, and therefore less hypotheses are generated. This is explained in more detail on the Background (page 105) page.

From a user perspective, search speed is linear to the maximum stack size. Compare the following system runs with stack size 1000, 100 (the default), 10, and 1:

```
% echo 'das ist ein kleines haus' | moses -f moses.ini -v 2 -s 1000
[...]
total hypotheses generated = 453
         number recombined = 69
```

```
            number pruned = 0
    number discarded early = 272
[...]
BEST TRANSLATION: this is a small house [11111]  [total=-28.923]

% echo 'das ist ein kleines haus' | moses -f moses.ini -v 2 -s 100
[...]
total hypotheses generated = 453
         number recombined = 69
             number pruned = 0
    number discarded early = 272
[...]
BEST TRANSLATION: this is a small house [11111]  [total=-28.923]

% echo 'das ist ein kleines haus' | moses -f moses.ini -v 2 -s 10
[...]
total hypotheses generated = 208
         number recombined = 23
             number pruned = 42
    number discarded early = 103
[...]
BEST TRANSLATION: this is a small house [11111]  [total=-28.923]

% echo 'das ist ein kleines haus' | moses -f moses.ini -v 2 -s 1
[...]
total hypotheses generated = 29
         number recombined = 0
             number pruned = 4
    number discarded early = 19
[...]
BEST TRANSLATION: this is a little house [11111]  [total=-30.991]
```

Note that the number of hypothesis entered on stacks is getting smaller with the stack size: 453, 453, 208, and 29.

As we have previously described with translation table pruning, we may also want to use the relative scores of hypothesis for pruning instead of a fixed limit. The two strategies are also called histogram pruning and threshold pruning.

Here some experiments to show the effects of different stack size limits and beam size limits.

```
% echo 'das ist ein kleines haus' | moses -f moses.ini -v 2 -s 100 -b 0
[...]
total hypotheses generated = 1073
         number recombined = 720
             number pruned = 73
    number discarded early = 0
[...]
```

```
% echo 'das ist ein kleines haus' | moses -f moses.ini -v 2 -s 1000 -b 0
[...]
total hypotheses generated = 1352
         number recombined = 985
             number pruned = 0
    number discarded early = 0
[...]

% echo 'das ist ein kleines haus' | moses -f moses.ini -v 2 -s 1000 -b 0.1
[...]
total hypotheses generated = 45
         number recombined = 3
             number pruned = 0
    number discarded early = 32
[...]
```

In the second example no pruning takes place, which means an exhaustive search is performed. With small stack sizes or small thresholds we risk search errors, meaning the generation of translations that score worse than the best translation according to the model.

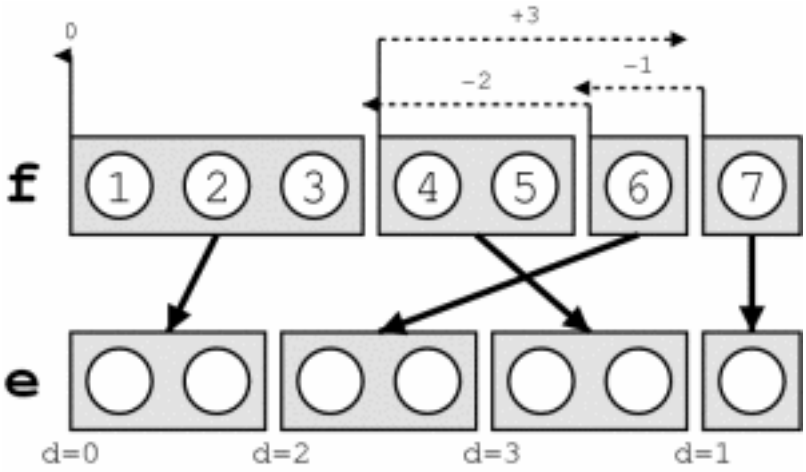
In this toy example, a worse translation is only generated with a stack size of 1. Again, by worse translation, we mean worse scoring according to our model (-30.991 vs. -28.923). If it is actually a worse translation in terms of translation quality, is another question. However, the task of the decoder is to find the best scoring translation. If worse scoring translations are of better quality, then this is a problem of the model, and should be resolved by better modeling.

2.1.8 Limit on Distortion (Reordering)

The basic reordering model implemented in the decoder is fairly weak. Reordering cost is measured by the number of words skipped when foreign phrases are picked out of order.

Total reordering cost is computed by D(e,f) = - Σ_i (d_i) where d for each phrase i is defined as d = abs(last word position of previously translated phrase + 1 - first word position of newly translated phrase).

This is illustrated by the following graph:



This reordering model is suitable for local reorderings: they are discouraged, but may occur with sufficient support from the language model. But large-scale reorderings are often arbitrary and effect translation performance negatively.

By limiting reordering, we can not only speed up the decoder, often translation performance is increased. Reordering can be limited to a maximum number of words skipped (maximum d) with the switch `-distortion-limit`, or short `-dl`.

Setting this parameter to 0 means monotone translation (no reordering). If you want to allow unlimited reordering, use the value -1.

2.2 Tutorial for Using Factored Models

Note: There may be some discrepencies between this description and the actual workings of the training script.

- Train a unfactored model (page 28)
- Train a model with POS tags (page 28)
- Train a model with generation and translation steps (page 31)
- Train a morphological analysis and generation model (page 32)
- Train a model with multiple decoding paths (page 33)

To work through this tutorial, you first need to have the data in place. The instructions also assume that you have the training script and the decoder in you executable path.

You can obtain the data as follows:

- `wget` http://www.statmt.org/moses/download/factored-corpus.tgz
- `tar xzf factored-corpus.tgz`

For more information on the training script, check the documentation, which is linked to on the right navigation column under "Training".

2.2.1 Train a unfactored model

The corpus package contains language models and parallel corpora with POS and lemma factors. Before playing with factored models, let us start with training a traditional phrase-based model:

```
% train-factored-phrase-model.perl \
   --corpus factored-corpus/proj-syndicate \
   --root-dir unfactored \
   --f de --e en \
   --lm 0:3:factored-corpus/surface.lm:0
```

This creates a phrase-based model in the directory `unfactore/model` in about 1 hour (?). For a quicker training run that only takes a few minutes (with much worse results) use `factored-corpus/proj-syn` which contains the first 1000 sentence pairs of the corpus.

```
% train-factored-phrase-model.perl \
   --corpus factored-corpus/proj-syndicate.1000 \
   --root-dir unfactored \
   --f de --e en \
   --lm 0:3:factored-corpus/surface.lm:0
```

This creates a typical phrase-based model, as specified in the created configuration file `moses.ini`. Here the part of the file that point to the phrase table:

```
[ttable-file]
0 0 5 /.../unfactored/model/phrase-table.0-0.gz
```

You can take a look at the generated phrase table, which starts as usual with rubbish but then occasionally contains some nice entries. The scores ensure that during decoding the good entries are preferred.

```
! ||| ! ||| 1 1 1 1 2.718
" ( ||| " ( ||| 1 0.856401 1 0.779352 2.718
" ) , ein neuer film ||| " a new film ||| 1 0.0038467 1 0.128157 2.718
" ) , ein neuer film über ||| " a new film about ||| 1 0.000831718 1 0.0170876 2.71
[...]
frage ||| issue ||| 0.25 0.285714 0.25 0.166667 2.718
frage ||| question ||| 0.75 0.555556 0.75 0.416667 2.718
```

2.2.2 Train a model with POS tags

Take a look at the training data. Each word is not only represented by its surface form (as you would expect in raw text), but also with additional factors.

```
% tail -n 1 factored-corpus/proj-syndicate.??
==> factored-corpus/proj-syndicate.de <==
korruption|korruption|nn|nn.fem.cas.sg floriert|florieren|vvfin|vvfin .|.|per|per

==> factored-corpus/proj-syndicate.en <==
corruption|corruption|nn flourishes|flourish|nns .|.|.
```

The German factors are

- surface form
- lemma
- part of speech
- part of speech with additional morphological information

The English factors are

- surface form
- lemma
- part of speech

Let us start simple and build a translation model that adds only the target part-of-speech factor on the output side:

```
% train-factored-phrase-model.perl \
   --corpus factored-corpus/proj-syndicate.1000 \
   --root-dir pos \
   --f de --e en \
   --lm 0:3:factored-corpus/surface.lm:0 \
   --lm 2:3:factored-corpus/pos.lm:0 \
   --translation-factors 0-0,2
```

Here, we specify with `-translation-factors 0-0,2` that the input factor for the translation table is the (0) surface form, and the output factor is (0) surface form and (2) part of speech.

```
[ttable-file]
0 0,2 5 /.../pos/model/phrase-table.0-0,2.gz
```

The resulting phrase table looks very similar, but now also contains part-of-speech tags on the English side:

```
! ||| !|. ||| 1 1 1 1 2.718
" ( ||| "|" (|( ||| 1 0.856401 1 0.779352 2.718
" ) , ein neuer film ||| "|" a|dt new|jj film|nn ||| 1 0.00403191 1 0.128157 2.718
" ) , ein neuer film über ||| "|" a|dt new|jj film|nn about|in ||| 1 0.000871765 1 0.017087
[...]
frage ||| issue|nn ||| 0.25 0.285714 0.25 0.166667 2.718
frage ||| question|nn ||| 0.75 0.625 0.75 0.416667 2.718
```

We also specified two language models. Besides the regular language model based on surface forms, we have a second language model that is trained on POS tags. In the configuration file this is indicated by two lines in the LM section:

```
[lmodel-file]
0 0 3 /.../factored-corpus/surface.lm
0 2 3 /.../factored-corpus/pos.lm
```

Also, two language model weights are specified:

```
# language model weights
[weight-l]
0.2500
0.2500
```

The part-of-speech language model includes preferences such as that determiner-adjective is likely followed by a noun, and less likely by a determiner:

```
-0.192859       dt jj nn
-2.952967       dt jj dt
```

This model can be used just like normal phrase based models:

```
% echo 'putin beschreibt menschen .' > in
% moses -f pos/model/moses.ini < in
[...]
BEST TRANSLATION: putin|nnp describes|vbz people|nns .|. [1111]  [total=-6.049]
<<0.000, -4.000, 0.000, -29.403, -11.731, -0.589, -1.303, -0.379, -0.556, 4.000>>
[...]
```

During the decoding process, not only words (`putin`), but also part-of-speech are generated (`nnp`).

Let's take a look what happens, if we input a German sentence that starts with the object:

```
% echo 'menschen beschreibt putin .' > in
% moses -f pos/model/moses.ini < in
BEST TRANSLATION: people|nns describes|vbz putin|nnp .|. [1111]  [total=-8.030]
<<0.000, -4.000, 0.000, -31.289, -17.770, -0.589, -1.303, -0.379, -0.556, 4.000>>
```

Now, this is not a very good translation. The model's aversion to do reordering trumps our ability to come up with a good translation. If we downweight the reordering model, we get a better translation:

```
% moses -f pos/model/moses.ini < in -d 0.2
BEST TRANSLATION: putin|nnp describes|vbz people|nns .|. [1111]  [total=-7.649]
<<-8.000, -4.000, 0.000, -29.403, -11.731, -0.589, -1.303, -0.379, -0.556, 4.000>>
```

Note that this better translation is mostly driven by the part-of-speech language model, which prefers the sequence `nnp vbz nns .` (-11.731) over the sequence `nns vbz nnp .` (-17.770). The surface form language model only shows a slight preference (-29.403 vs. -31.289). This is because these words have not been seen next to each other before, so the language model has very little to work with. The part-of-speech language model is aware of the count of the nouns involved and prefers a singular noun before a singular verb (`nnp vbz`) over a plural noun before a singluar verb (`nns vbz`).

To drive this point home, the unfactored model is not able to find the right translation, even with downweighted reordering model:

```
% moses -f unfactored/model/moses.ini < in -d 0.2
people describes putin . [1111]  [total=-11.410]
<<0.000, -4.000, 0.000, -31.289, -0.589, -1.303, -0.379, -0.556, 4.000>>
```

2.2.3 Train a model with generation and translation steps

Let us now train a slightly different factored model with the same factors. Instead of mapping from the German input surface form directly to the English output surface form and part of speech, we now break this up into two mapping steps, one translation step that maps surface forms to surface forms, and a second step that generates the part of speech from the surface form on the output side:

```
% train-factored-phrase-model.perl \
    --corpus factored-corpus/proj-syndicate.1000 \
    --root-dir pos-decomposed \
    --f de --e en \
    --lm 0:3:factored-corpus/surface.lm:0 \
    --lm 2:3:factored-corpus/pos.lm:0 \
    --translation-factors 0-0 \
    --generation-factors 0-2 \
    --decoding-steps t0,g0
```

Now, the translation step is specified only between surface forms (`-translation-factors 0-0`) and a generation step is specified (`-generation-factors 0-2`), mapping (0) surface form to (2) part of speech. We also need to specified in which order the mapping steps are applied (`-decoding-steps t0,g0`).

Besides the phrase table that has the same format as the unfactored phrase table, we now also have a generation table. It is referenced in the configuration file:

```
[...]
# generation models: source-factors, target-factors, number-of-weights, filename
[generation-file]
0 2 2 /.../pos-decomposed/model/generation.0-2
[...]
[weight-generation]
0.3
0
[...]
```

Let us take a look at the generation table:

```
% more pos-decomposed/model/generation.0-2
nigerian nnp 1.0000000  0.0008163
proven vbn 1.0000000  0.0021142
issue nn 1.0000000  0.0021591
[...]
control vb 0.1666667  0.0014451
control nn 0.8333333  0.0017992
[...]
```

The beginning is not very interesting. As most words, `nigerian`, `proven`, and `issue` occur only with one part of speech, e.g., p(`nnp`|`nigerian`) = `1.0000000`. `Some words, however, such as controloccur with multiple part of speech, such as base form verb (vb) and single noun (nn@@)`.

The table also contains the reverse translation probability p(`nigerian`|`nnp`) = 0.0008163. In our example, this may not be a very useful feature, it basically hurts open class words, especially unusual ones. If we do not want this feature, we can also train the generation model as single-featured by the switch `-generation-type single`.

2.2.4 Train a morphological analysis and generation model

Translating surface forms seems to be a somewhat questionable persuit. It does not seem to make much sense to treat different word forms of the same lemma, such as `mensch` and `menschen` differently. In the worst case, we will have seen only one of the word forms, so we are not able to translate the other. This is what in fact happens in this example:

```
% echo 'ein mensch beschreibt putin .' > in
% moses.1430.srilm -f unfactored/model/moses.ini < in
a mensch|UNK|UNK|UNK describes putin . [11111]  [total=-158.818]
<<0.000, -5.000, -100.000, -127.565, -1.350, -1.871, -0.301, -0.652, 4.000>>
```

Factored translation models allow us to create models that do morphological analysis and decomposition during the translation process. Let us now train such a model:

```
% train-factored-phrase-model.perl \
   --corpus factored-corpus/proj-syndicate.1000 \
   --root-dir morphgen \
   --f de --e en \
   --lm 0:3:factored-corpus/surface.lm:0 \
   --lm 2:3:factored-corpus/pos.lm:0 \
   --translation-factors 1-1+3-2 \
   --generation-factors 1-2+1,2-0 \
   --decoding-steps t0,g0,t1,g1 \
```

We have a total of four mapping steps:

- a translation step that maps lemmas (`1-1`)
- a generation steps that sets possible part-of-speech tags for a lemma (`1-2`)
- a translation step that maps morphological information to part-of-speech tags (`3-2`)
- a generation step that maps part-of-speech tag and lemma to a surface form (`1,2-0`).

This enables us now to translate the sentence above:

```
% echo 'ein|ein|art|art.indef.z mensch|mensch|nn|nn.masc.nom.sg beschreibte|beschreiben|vvfin
% moses -f morphgen/model/moses.ini < in
BEST TRANSLATION: a|a|dt individual|individual|nn describes|describe|vbz putin|putin|nnp .|.|
<<0.000, -5.000, 0.000, -38.631, -13.357, -2.773, -21.024, 0.000, -1.386, -1.796, -4.341, -3.
```

Note that this is only possible, because we have seen an appropriate word form in the output language. The word `individual` occurs as single noun in the parallel corpus, as translation of `einzelnen`. To overcome this limitation, we may train generation models on large monolingual corpora, where we expect to see all possible word forms.

2.2.5 Train a model with multiple decoding paths

Decomposing translation into a process of morphological analysis and generation will make our translation model more robust. However, if we have seen a phrase of surface forms before, it may be better to take advantage of such rich evidence.

The above model poorly translates sentences as it does use the source surface form at all, relying on translating the properties of the surface forms.

In practice, we fair better when we allow both ways to translate in parallel. Such a model is trained by the introduction of decoding paths. In our example, one decoding path is the

morphological analysis and generation as above, the other path the direct mapping of surface forms to surface forms (and part-of-speech tags, since we are using a part-of-speech tag language model):

```
% train-factored-phrase-model.perl \
    --corpus factored-corpus/proj-syndicate.1000 \
    --root-dir morphgen-backoff \
    --f de --e en \
    --lm 0:3:factored-corpus/surface.lm:0 \
    --lm 2:3:factored-corpus/pos.lm:0 \
    --translation-factors 1-1+3-2+0-0,2 \
    --generation-factors 1-2+1,2-0 \
    --decoding-steps t0,g0,t1,g1:t2
```

This command is almost identical to the previous training run, except for the additional translation table `0-0,2` and its inclusion as a different decoding path `:t2`.

A strategy for translating surface forms which have not been seen in the training corpus is to translate it's lemma instead. This is especially useful for translation from morphologically rich languages to simpler languages, such as German to English translation.

```
% train-factored-phrase-model.perl \
    --corpus factored-corpus/proj-syndicate.1000 \
    --root-dir lemma-backoff \
    --f de --e en \
    --lm 0:3:factored-corpus/surface.lm:0 \
    --lm 2:3:factored-corpus/pos.lm:0 \
    --translation-factors 0-0,2+1-0,2 \
    --decoding-steps t0:t1
```

2.3 Syntax Tutorial

<dl><dd><dl><dd><dl><dd><dl><dd><dl><dd><dl><dd>

23 And when they came to Marah, they could not drink of the waters of Marah, for they were bitter: therefore the name of it was called Marah.

24 And the people murmured against Moses, saying, What shall we drink?

25 And he cried unto the Lord; and the Lord showed him a **tree***, which when he had cast into the waters, the waters were made sweet: there he made for them a statute and an ordinance, and there he proved them.*

Exodus 15, 23-25

</dd></dl></dd></dl></dd></dl></dd></dl></dd></dl></dd></dl> Moses supports models that have become known as *hierarchical phrase-based models* and *syntax-based models*. In the following, we refer to these models as **tree-based models**. The implementation in Moses is currently in a beta-release status in a separate branch.

2.3.1 Tree-Based Models

Traditional phrase based models have as atomic translation step the mapping of an input phrase to an output phrase. Tree-based models operate on so-called grammar rules, which include variables in the mapping rules:

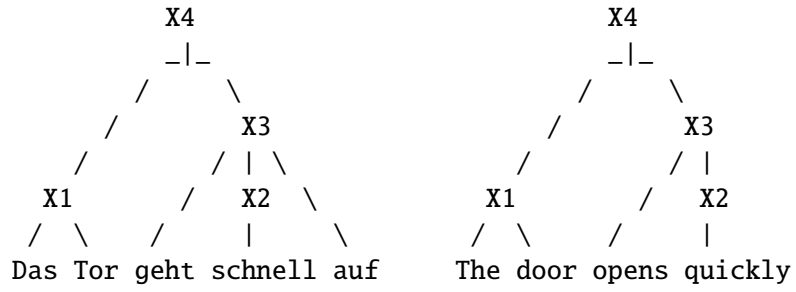
```
{\tt ne $\text{X}_\text{1}$ pas -> not $\text{X}_\text{1}$} (French-English)
{\tt ate $\text{X}_\text{1}$ -> habe $\text{X}_\text{1}$ gegessen} (English-German)
{\tt $\text{X}_\text{1}$ of the $\text{X}_\text{2}$ -> le $\text{X}_\text{2}$ $\text{X}_\text{1}$ de
```

The variables in these grammar rules are called non-terminals, since they occurrence indicates that the process has not yet terminated to produce the final words (the *terminals*). Besides a generic non-terminal `X`, linguistically motivated non-terminals such as `NP` (noun phrase) or `VP` (verb phrase) may be used as well in a grammar (or translation rule set).

We call these models tree-based, because during the translation is formed in data structure is created that is a called a tree. To fully make this point, consider the following input and translation rules:

```
Input: {\tt Das Tor geht schnell auf}
Rules: {\tt Das Tor -> The door}
       {\tt schnell -> quickly}
       {\tt geht $\text{X}_\text{1}$ auf -> opens $\text{X}_\text{1}$}
       {\tt $\text{X}_\text{1}$ $\text{X}_\text{2}$ -> $\text{X}_\text{1}$ $\text{X}_\text{2}$
```

When applying these rules in the given order, we produce the translation *The door opens quickly* in the following fashion:



First the simple phrase mappings (1) `Das Tor` to `The door` and (2) `schnell` to `quickly` are carried out. This allows for the application of the more complex rule (3) `geht` X_1 `auf` to `opens` X_1. Note that at this point, the non-terminal `X`, which covers the input span over `schnell` is replaced by a known translation `quickly`. Finally, the glue rule (4) X_1 X_2 to X_1 X_2 combines the two fragments into a complete sentence.

Here is how the spans over the input words are getting filled in:

```
|4 ----  The door opens quickly ---- |
```

```
|           |3 --- opens quickly --- |
|1 The door |      |2 quickly |      |
| Das | Tor | geht | schnell  | auf  |
```

Formally, such context free grammars are more constraint than the formalism for phrase-based models. In practice, however, phrase-based models use a reordering limit, which leads to linear decoding time. For tree-based models, decoding is not linear with respect to sentence length, unless reordering limits are used.

Current research in tree-based models has the expectation to build translation models that more closely model the underlying linguistic structure of language, and its essential element: recursion. This is an active field of research.

A Word on Terminology

You may have read in the literature about hierarchical phrase-based, string-to-tree, tree-to-string, tree-to-tree, target-syntactified, syntax-augmented, syntax-directed, syntax-based, grammar-based, etc., models in statistical machine translation. What do the tree-based models support? All of the above.

The avalanche of terminology stems partly from the need of researchers to carve out their own niche, partly from the fact is that work in this area has not yet fully settled on a agreed framework, but also from a fundamental difference. As we already pointed out, the motivation for tree-based models are linguistic theories and their syntax trees. So, when we build a data structure called a *tree* (as Computer Scientist call it), do we mean that we build a linguistic syntax *tree* (as Linguists call it)?

Not always, and hence the confusion. In all our examples above we used a single non-terminal `X`, so not many will claim the the result is a proper linguistic syntax with its noun phrases `NP`, verb phrases `VP`, and so on. To distinguish models that use proper linguistic syntax on the input side, on the output side, on both, or on neither all this terminology has been invented.

Let's decipher common terms found in the literature:

- hierarchical phrase-based: no linguistic syntax
- string-to-tree: linguistic syntax only in output language
- tree-to-string: linguistic syntax only in input language
- tree-to-tree: linguistic syntax in both languages
- target-syntactified: linguistic syntax only in output language
- syntax-augmented: linguistic syntax only in output language
- syntax-directed: linguistic syntax only in input language

- syntax-based: unclear, we use it for models that have any linguistic syntax
- grammar-based: wait, what?

In this tutorial, we refer to un-annotated trees as **trees**, and to trees with syntactic annotation as **syntax**. So a so-called string-to-tree model is here called a target syntax model.

Chart Decoding

Phrase-Based decoding generates a sentence from left to right, by adding phrases to the end of a partial translation. Tree-based decoding builds a chart, which consists of partial translation for all possible spans over the input sentence.

Currently Moses implements a CKY+ algorithm for arbitrary number of non-terminals per rule and an arbitrary number of types of non-terminals in the grammar.

2.3.2 Decoding

We assume that you have already installed the chart decoder, as described in the Get Started[1] section.

You can find an example model for the decoder from the Moses web site[2]. Unpack the tar ball and enter the directory `sample-models`:

```
% wget://www.statmt.org/moses/download/sample-models.tgz
% tar xzf sample-models.tgz
% cd sample-models
```

The decoder is called just as for phrase models:

```
% echo 'das ist ein haus' | moses_chart -f syntax-model/moses.ini > out
% cat out
this is a house
```

What happened here?

Trace

Using the open `-T` we can some insight how the translation was assembled:

```
41 X TOP -> <s> S </s> (1,1)             [0..5] -3.593 <<0.000, -2.606, -9.711, 2.526>> 20
20 X S   -> NP V NP    (0,0) (1,1) (2,2) [1..4] -1.988 <<0.000, -1.737, -6.501, 2.526>> 3 5 1
 3 X NP  -> this                         [1..1]  0.486 <<0.000, -0.434, -1.330, 2.303>>
 5 X V   -> is                           [2..2] -1.267 <<0.000, -0.434, -2.533, 0.000>>
11 X NP  -> DT NN      (0,0) (1,1)       [3..4] -2.698 <<0.000, -0.869, -5.396, 0.000>> 7 9
 7 X DT  -> a                            [3..3] -1.012 <<0.000, -0.434, -2.024, 0.000>>
 9 X NN  -> house                        [4..4] -2.887 <<0.000, -0.434, -5.774, 0.000>>
```

[1] http://www.statmt.org/moses/?n=Development.GetStarted#chart
[2] http://www.statmt.org/moses/download/sample-models.tgz

Each line represents a hypothesis that is part of the derivation of the best translation. The pieces of information in each line (with the first line as example) are:

- the hypothesis number, a sequential identifier (41)
- the input non-terminal (X)
- the output non-termial (S)
- the rule used to generate this hypothesus (TOP -> <s> S </s>)
- alignment information between input and output non-terminals in the rule ((1,1))
- the span covered by the hypothesis, as defined by input word positions ([0..5])
- the score of the hypothesis (3.593)
- its component scores (<<...>>):
  - unknown word penalty (0.000)
  - word penalty (-2.606)
  - language model score (-9.711)
  - rule application probability (2.526)
- prior hypotheses, i.e. the children nodes in the tree, that this hypothesis is built on (20)

As you can see, the model used here is a target-syntax model, It uses linguistic syntactic annotation on the target side, but on the input side everything is labeled X.

Rule Table

If we look at the syntax-model directory, we find two files: the configuration file moses.ini which points to the language model (in lm/europarl.srilm.gz) and the rule table, and the rule table file rule-table.

The configuration file moses.ini has a fairly familiar format. It is mostly identical to the configuration file for phrase-based models. We will describe further below in detail the new parameters of the chart decoder.

The rule table rule-table also looks familiar to anybody who has seen a phrase table of a phrase-based models. Here some lines as example:

```
[X] [NP] ||| das ||| this |||  ||| 0.1
[X] [VP] ||| macht [X] auf ||| opens [NP] ||| 1-1 ||| 0.8
[X] [NP] ||| [X] [X] ||| [DT] [NN] ||| 0-0 1-1 ||| 1.0
[X] [TOP] ||| <s> [X] </s> ||| <s> [S] </s> ||| 1-1 ||| 1.0
```

Each line in the rule table describes one translation rule. It consists of five components separated by three bars:

- the input and output non-terminal
- the input half of the right-hand-side of the rule
- the output half of the right-hand-side of the rule
- the alignment between non-terminals (using word positions)
- score(s): here only one, but typically multiple scores are used

In the excerpt above, we have quite different rules. Let us look at them one-by-one:

```
[X] [NP] ||| das ||| this |||  ||| 0.1
```

The first rule is a pretty basic lexical translation rule, that in addition to translating the input word `das` into the output word `this` also labels the resulting translation as `NP`:

```
[X] [VP] ||| macht [X] auf ||| opens [NP] ||| 1-1 ||| 0.8
```

The second rule is a more complex rule that does lexical mapping, but also contains a non-terminal — labeled `NP` on the output side. The rule is `macht` $\texttt{X}_1$ `auf -> opens` $\texttt{NP}_1$, and the resulting partial translation is labeled as verb phrase (`VP`).

```
[X] [NP] ||| [X] [X] ||| [DT] [NN] ||| 0-0 1-1 ||| 1.0
```

The next rule does not map any words at all. It does allow the combination of on output span labeled as determiner (`DT`) and an output span labeled as noun (`NN`) into a noun phrase (`NP`).

```
[X] [TOP] ||| <s> [X] </s> ||| <s> [S] </s> ||| 1-1 ||| 1.0
```

Finally, this rather technical rule applies only to spans that cover everything except the sentence boundary markers <s> and </s>. It completes a translation with of a sentence span (`S`).

More Example

The second rule in the table, that we just glanced at, allows something quite interesting: the translation of a non-contiguous phrase: `macht X auf`.

Let us try this with the decoder on an example sentence:

```
% echo 'er macht das tor auf' | moses_chart -f syntax-model/moses.ini -T
[...]
14 X TOP -> <s> S </s> (1,1)       [0..6] -7.833 <<0.000, -2.606, -17.163, 1.496>> 13
13 X S   -> NP VP      (0,0) (1,1) [1..5] -6.367 <<0.000, -1.737, -14.229, 1.496>> 2 11
 2 X NP  -> he                     [1..1] -1.064 <<0.000, -0.434,  -2.484, 0.357>>
```

```
11 X VP  -> opens NP   (1,1)       [2..5] -5.627 <<0.000, -1.303, -12.394, 1.139>> 10
10 X NP  -> DT NN      (0,0) (1,1) [3..4] -3.154 <<0.000, -0.869,  -7.224, 0.916>> 6 7
 6 X DT  -> the                    [3..3]  0.016 <<0.000, -0.434,  -0.884, 0.916>>
 7 X NN  -> gate                   [4..4] -3.588 <<0.000, -0.434,  -7.176, 0.000>>
he opens the gate
```

You see the creation application of the rule in the creation of hypothesis `11`. It generates `opens NP` to cover the input span `[2..5]` by using hypothesis `10`, which covered the span `[3..4]`.

Note that this rule allows us to do something that is not possible with a simple phrase-based model. Phrase-based models in Moses require that all phrases are contiguous, they can not have gaps.

The final example illustrates how reordering works in a tree-based model:

```
% echo 'ein haus ist das' | moses_chart -f syntax-model/moses.ini -T
41 X TOP -> <s> S </s> (1,1)             [0..5] -2.900 <<0.000, -2.606, -9.711, 3.912>> 18
18 X S   -> NP V NP    (0,2) (1,1) (2,0) [1..4] -1.295 <<0.000, -1.737, -6.501, 3.912>> 11
11 X NP  -> DT NN      (0,0) (1,1)       [1..2] -2.698 <<0.000, -0.869, -5.396, 0.000>> 2 4
 2 X DT  -> a                            [1..1] -1.012 <<0.000, -0.434, -2.024, 0.000>>
 4 X NN  -> house                        [2..2] -2.887 <<0.000, -0.434, -5.774, 0.000>>
 5 X V   -> is                           [3..3] -1.267 <<0.000, -0.434, -2.533, 0.000>>
 8 X NP  -> this                         [4..4]  0.486 <<0.000, -0.434, -1.330, 2.303>>
this is a house
```

The reordering in the sentence happens when hypothesis `18` is generated. The non-lexical rule `S -> NP V NP` takes the underlying children nodes in inverse order (`(0,2) (1,1) (2,0)`).

Not any arbitrary reordering is allowed — as is the case in in phrase models. Reordering has to be motivated by a translation rule. If the model uses real syntax, there has to be a syntactic justification for the reordering.

2.3.3 Decoder Parameters

The most important consideration in decoding is a speed/quality trade-off. If you want to win competitions, you want the best quality possible, even if it takes a week to translate 2000 sentences. If you want to provide an online service, you know that users get impatient, when they have to wait more than a second.

Beam Settings

The chart decoder has a implementation of CKY decoding using cube pruning. The latter means that only a fixed number of hypotheses are generated for each span. This number can be changed with the option `cube-pruning-pop-limit` (or short `cbp`). The default is 1000, higher numbers slow down the decoder, may result in better quality.

Another setting that directly affects speed is the number of rules that are considered for each input left hand side. It can be set with `ttable-limit`.

Limiting Reordering

The number of spans that are filled during chart decoding is quadratic with respect to sentence length. But it gets worse. The number of spans that be combined into into a span grows linear with sentence length for binary rules, quadratic for trinary rules, and so on. In short, long sentences become a problem. A drastic solution is the size of internal spans to a maximum number.

This sounds a bit extreme, but does make some sense for non-syntactic models. Reordering is limited in phrase-based models, and non-syntactic tree-based models (better known as hierarchical phrase-based models) should limit reordering for the same reason: they are just not very good at long-distance reordering anyway.

The limit on span sizes can be set with `max-chart-span`. In fact its default is `10`, which is not a useful setting for syntax models.

Handling Unknown Words

In a target-syntax model, unknown words that just copied verbatim into the output need to get a non-terminal label. In practice unknown words tend to be open class words, most likely names, nouns, or numbers. With the option `unknown-lhs` you can specify a file that contains pairs of non-terminal labels and their probability per line.

Technical Settings

The parameter `non-terminals` us used to specified privileged non-terminals. These are used for unknown words (unless there is a unknown word label file) and to define the non-terminal label for on the input side, when this is not specified.

Typically, we want to consider all possible rules that apply. However, with a large maximum phrase length, too many rule tables and no rule table limit, this may explode. The number of rules consider can be limited with `rule-limit`. Default is 5000.

2.3.4 Training

In short, training uses the identical training script as phrase-based models. When running `train-model.perl`, you will have to specify additional parameters `-hierarchical -glue-grammar`. You typically will also reduce the number of lexical items in the grammar with `-max-phrase-length 5`.

That's it.

Training Parameters

There are a number of additional decisions about the type of rules you may want to include in your model. This is typically a size / quality trade-off: Allowing more rule types increases

the size of the rule table, but lead to better results. Bigger rule tables have a negative impact on memory use and speed of the decoder.

There are two parts to creating a rule table: the extraction of rules and the scoring of rules. The first can be modified with the parameter `-extract-options="..."` of `train-model.perl`. The second with `-score-options="..."`.

Here are the extract options:

- `-OnlyDirect`: only create a model with direct conditional probabilities *p(f|e)* instead of the default direct and indirect (*p(f|e)* and *p(e|f)*).
- `-MaxSpan SIZE`: maximum span size of the rule. Default is `15`.
- `-MaxSymbolsSource SIZE` and `-MaxSymbolsTarget SIZE`: While a rule may be extracted from a large span, much of it may be knocked out by sub-phrases that are substituted by non-terminals. So, fewer actual symbols (non-terminals and words remain). The default maximum number of symbols is `5` for the source side, and practically unlimited (`999`) for the target side.
- `-MinWords SIZE`: minimum number of words in a rule. Default is `1`, meaning that each rule has to have at least one word in it. If you want to allow non-lexical rules set this to zero. You will not want to do this for hierarchical models.
- `-AllowOnlyUnalignedWords`: this is related to the above. A rule may have words in it, but these may be unaligned words that are not connected. By default, at least one aligned word is required. Using this option, this requirement is dropped.
- `-MaxNonTerm SIZE`: the number of non-terminals on the right hand side of the rule. This has an effect on the arity of rules, in terms of non-terminals. Default is to generate only binary rules, so the setting is `2`.
- `-MinHoleSource SIZE` and `-MinHoleTarget SIZE`: When sub-phrases are replaced by non-terminals, we may require a minimum size for these subphrases. The default is `2` on the source side and `1` (no limit) on the target side.
- `-DisallowNonTermConsecTarget` and `-NonTermConsecSource`. We may want to restrict if their can be neighboring non-terminals in rules. In hierarchical models there is a bad effect on decoding to allow neighboring non-terminals on the source side. The default is to disallow this – it is allowed on the target side. These switches override the defaults.
- `-NoFractionalCounting`: For any given source span, any number of rules can be generated. By default, fractional counts are assigned, so probability of these rules adds up to one. This options leads to the count of one for each rule.
- `-NoNonTermFirstWord`: Disallow that a rule starts with a non-terminal.

Once rules are collected, the file of rules and their counts has to be converted into a probabilistic model. This is called rule scoring, and there are also some additional options:

- `-OnlyDirect`: only estimate direct conditional probabilities. Note that this option needs to be specified for both rule extraction and rule scoring.
- `-NoLex`: only include rule-level conditional probabilities, not lexical scores.
- `-GoodTuring`: Use Good Turing discounting to reduce actual accounts. This is a good thing, use it.

Training Syntax Models

Training hierarchical phrase models, i.e., tree-based models without syntactic annotation, is pretty straight-forward. Adding syntactic labels to rules, either on the source side or the target side, is not much more complex. The main hurdle is to get the annotation. This requires a syntactic parser.

Syntactic annotation is provided by annotating all the training data (input or output side, or both) with syntactic labels. The format that is used for this uses XML markup. Here an example:

```
<tree label="NP"> <tree label="DET"> {\bf the} </tree> <tree label="NN"> {\bf cat} </tree> </
```

So, constituents are surrounded by a opening and a closing <tree> tag, and the label is provided with the parameter `label`. The XML markup also allows for the placements of the tags in other positions, as long as a `span` parameter is provided:

```
<tree label="NP" span="0-1"/> <tree label="DET" span="0-0"/> <tree label="DET" span="1-1"/> {
```

After annotating the training data with syntactic information, you can simply run `train-model.perl` as before, except that the switches `-source-syntax` or `-target-syntax` (or both) have to be set.

You may also change some of the extraction settings, for instance `-MaxSpan 999`.

Annotation Wrappers

To obtain the syntactic annotation, you will likely use a third party parser, which has its own idiosyncratic input and output format. You will need to write a wrapper script that converts it into the Moses format for syntax trees.

We provide wrappers (in `scripts/training/wrapper`) for the following parsers.

- **Bitpar** is available from the web site of the University of Stuttgart[3]. The wrapper is `parse-de-bitpar.perl`

[3] http://www.ims.uni-stuttgart.de/tcl/SOFTWARE/BitPar.html

- **Collins parser** is availble from MIT[4]. The wrapper is `parse-de-collins.perl`

If you wrote your own wrapper for a publicly available parsers, please share it with us!

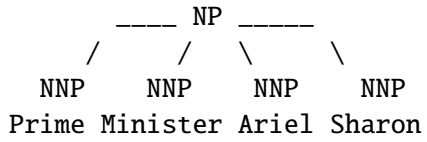
Relaxing Parses

The use of syntactic annotation puts severe constraints on the number of rules that can be extracted, since each non-terminal has to correspond to an actual non-terminal in the syntax tree.

Recent research has proposed a number of relaxations of this constraint. The program `relax-parse` (in `training/phrase-extract`) implements two kinds of parse relaxations: binarization and a method proposed under the label of syntax-augmented machine translation (SAMT) by Zollmann and Venugopal.

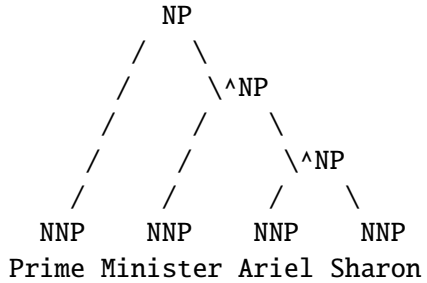
Readers familiar with the concept of binarizing grammars in parsing, be warned: We are talking here about modifying parse trees, which changes the power of the extracted grammar, not binarization as a optimization step during decoding.

The idea is the following: If the training data contains a subtree such as



then it is not possible to extract translation rules for `Ariel Sharon` without additional syntactic context. Recall that each rule has to match a syntactic constituent.

The idea of relaxing the parse trees is to add additional internal nodes that makes the extraction of additional rules possible. For instance left-binarization adds two additional nodes and converts the subtree into:



The additional node with the label N̂P allows for the straight-forward extraction of a translation rule (of course, unless the word alignment does not provide a consistent alignment).

The program `relax-parse` allows the following tree transformations:

[4] http://people.csail.mit.edu/mcollins/code.html

- `-LeftBinarize` and `-RightBinarize`: adds internal nodes as in the example above. Right-binarization creates a right-branching tree.
- `-SAMT 1`: combines pairs of neighboring children nodes, into tags such as `DET+ADJ`. Also nodes for everything except the first child (`NP`
  `DET`) and everything except the last child (`NP/NN`) are added.
- `-SAMT 2`: combines any pairs of neighboring nodes, not only children nodes, e.g., `VP+DET`.
- `-SAMT 3`: not implemented.
- `-SAMT 4`: as above, but in addition each previously unlabeled node is labeled as `FAIL`, so no syntactic constraint on grammar constraint remains.

Note that you can also use both `-LeftBinarize` and `-RightBinarize`. Note that in this case, as with all the SAMT relaxations, the resulting annotation is not any more a tree, since there is not a single set of rule applications that generates the structure (now called a forest).

Here an example, what parse relaxation does to the number of rules extracted (English-German News Commentary, using Bitpar for German, no English syntax):

Relaxation Setting	Number of Rules
no syntax	59,079,493
basic syntax	2,291,400
left-binarized	2,914,348
right-binarized	2,979,830
SAMT 1	8,669,942
SAMT 2	35,164,756
SAMT 4	131,889,855

On-Disk Rule Table

The rule table may become too big to fit into the RAM of the machine. Instead of loading the rules into memory, it is also possible to leave the rule table on disk, and retrieve rules on demand.

When choosing this option, you first need to convert the rule table into a binary prefix format. This is done with the command `CreateOnDiskPt` which is in the directory `CreateOnDiskPt/src`:

```
CreateOnDiskPt [#source factors] [#target factors] [#scores] [ttable-limit] [index of p(e|f)
```

There are currently still performance issues with creating such binarized phrase tables for very large rule tables (100,000,000s of rules).

2.4 Support Tools

2.4.1 Overview

Scripts are in the `scripts` subdirectory in the SVN release on Sourceforge.

The following basic tools are described elsewhere:

- Moses decoder (page 15)
- Training script `train-factored-phrase-model.perl` (page 77)
- Corpus preparation `clean-corpus-n.perl` (page 76)
- Minimum error rate training (tuning) `mert-moses-new.pl` (replacing the old version `mert-moses.pl`) (page 98)

2.4.2 Content

- Converting Pharaoh configuration files to Moses configuration files (page 46)
- Moses decoder in parallel (page 47)
- Filtering phrase tables for Moses (page 47)
- Reducing and Extending the Number of Factors (page 48)
- Scoring translations with BLEU (page 48)
- Missing and Extra N-Grams (page 48)
- Making a Full Local Clone of Moses Model + ini File (page 48)
- Absolutizing Paths in moses.ini (page 49)
- Printing Statistics about Model Components (page 49)
- Recaser (page 50)
- Truecaser (page 50)
- Searchgraph to DOT (page 51)

2.4.3 Converting Pharaoh configuration files to Moses configuration files

Moses is a successor the the Pharaoh decoder, so you can use the same models that work for Pharaoh and use them with Moses. The following script makes the necessary changes to the configuration file:

```
exodus.perl < pharaoh.ini > moses.ini
```

2.4.4 Moses decoder in parallel

Since decoding large amounts of text takes a long time, you may want to split up the text into blocks of a few hundred sentences (or less), and distribute the task across a Sun GridEngine cluster. This is supported by the script `moses-parallel.pl`, which is run as follows:

```
moses-parallel.pl -decoder decoder -config cfgfile -i input -jobs N [options]
```

Use absolute paths for your parameters (decoder, configuration file, models, etc.).

- `decoder` is the file location of the binary of Moses used for decoding
- `cfgfile` is the configuration fileofthe decoder
- `input` is the file to translate
- `N` is the number of processors you require
- `options` are used to overwrite parameters provided in `cfgfile` Among them, overwrite the following two parameters for nbest generation (NOTE: they differ from standard Moses)
  - `-n-best-file` output file for nbest list
  - `-n-best-size` size of nbest list

2.4.5 Filtering phrase tables for Moses

Phrase tables easily get too big, but for the translation of a specific set of text only a fraction of the table is needed. So, you may want to filter the translation table, and this is possible with the script:

```
filter-model-given-input.pl filter-dir config input-file
```

This creates a filtered translation table with new configuration file in the directory `filter-dir` from the model specified with the configuration file `config` (typically named `moses.ini`), given the (tokenized) input from the file `input-file`.

In the advanced feature section, you find the additional option of binarizing translation and reordering table, which allows these models to be kept on disk and queried by the decoder. If you want to both filter and binarize these tables, you can use the script:

```
filter-and-binarize-model-given-input.pl binarizer filter-dir config input-file
```

The additional `binarizer` option points to the appropriate version of `processPhraseTable`.

2.4.6 Reducing and Extending the Number of Factors

Instead of the two following scripts, this one does both at the same time, and is better suited for our directory structure and factor naming conventions:

```
reduce_combine.pl
   czeng05.cs
   0,2 pos lcstem4
   > czeng05_restricted_to_0,2_and_with_pos_and_lcstem4_added
```

2.4.7 Scoring translations with BLEU

A simple BLEU scoring tool is the script `multi-bleu.perl`:

```
multi-bleu.perl reference < mt-output
```

Reference file and system output have to be sentence-aligned (line X in the reference file corresponds to line X in the system output). If multiple reference translation exist, these have to be stored in seperate files and named `reference0`, `reference1`, `reference2`, etc. All the texts need to be tokenized.

A popular script to score translations with BLEU is the NIST mteval script[5]. It requires that text is wrapped into a SGML format. This format is used for instance by the NIST evaluation[6] and the WMT Shared Task evaluations[7]. See the latter for more details on using this script.

2.4.8 Missing and Extra N-Grams

Missing n-grams are those that all reference translations wanted but MT system did not produce. Extra n-grams are those that the MT system produced but none of the references approved.

```
missing_and_extra_ngrams.pl hypothesis reference1 reference2 ...
```

2.4.9 Making a Full Local Clone of Moses Model + ini File

Assume you have a `moses.ini` file already and want to run an experiment with it. Some months from now, you might still want to know what exactly did the model (incl. all the tables) look like, but people tend to move files around or just delete them.

To solve this problem, create a blank directory, go in there and run:

```
clone_moses_model.pl ../path/to/moses.ini
```

[5] http://www.nist.gov/speech/tests/mt/2008/scoring.html
[6] http://www.nist.gov/speech/tests/mt/2009/
[7] http://www.statmt.org/wmt09/translation-task.html

`close_moses_model.pl` will make a copy of the `moses.ini` file and local symlinks (and if possible also hardlinks, in case someone deleted the original file) to all the tables and language models needed.

It will be now safe to run moses locally in the fresh directory.

2.4.10 Absolutizing Paths in moses.ini

Run:

```
absolutize_moses_model.pl  ../path/to/moses.ini > moses.abs.ini
```

to build an ini file where all paths to model parts are absolute. (Also checks the existence of the files.)

2.4.11 Printing Statistics about Model Components

The script

```
analyse_moses_model.pl moses.ini
```

Prints basic statistics about all components mentioned in the moses.ini. This can be useful to set the order of mapping steps to avoid explosion of translation options or just to check that the model components are as big/detailed as we expect.

Sample output lists information about a model with 2 translation and 1 generation step. The three language models over three factors used and their n-gram counts (after discounting) are listed, too.

```
Translation 0 -> 1 (/fullpathto/phrase-table.0-1.gz):
  743193        phrases total
  1.20  phrases per source phrase
Translation 1 -> 2 (/fullpathto/phrase-table.1-2.gz):
  558046        phrases total
  2.75  phrases per source phrase
Generation 1,2 -> 0 (/fullpathto/generation.1,2-0.gz):
  1.04  outputs per source token
Language model over 0 (/fullpathto/lm.1.lm):
  1     2       3
  49469 245583  27497
Language model over 1 (/fullpathto/lm.2.lm):
  1     2       3
  25459 199852  32605
Language model over 2 (/fullpathto/lm.3.lm):
  1     2       3       4       5       6       7
  709   20946   39885   45753   27964   12962   7524
```

2.4.12 Recaser

Often, we train machine translation systems on lowercased data. If we want to present the output to a user, we need to re-case (or re-capitalize) the output. Moses provides a simple tool to recase data, which essentially runs Moses without reordering, using a word-to-word translation model and a cased language model.

The recaser requires a model (i.e., the word mapping model and language model mentioned above), which is trained with the command:

```
train-recaser.perl --dir MODEL --corpus CASED [--ngram-count NGRAM] [--train-script TRAIN]
```

The script expects a cased (but tokenized) training corpus in the file `CASED`, and creates a reordering model in the directory `MODEL`. Since the script needs to run SRILM's `ngram-count` and the Moses `train-factored-phrase-model.perl`, these need to be specified, otherwise the hard-coded defaults are used.

To recase output from the Moses decoder, you run the command

```
recase.perl --in IN --model MODEL/moses.ini --moses MOSES [--lang LANGUAGE] [--headline SGM
```

The input is in file `IN`, the output in file `OUT`. You also need to specify a recasing model `MODEL`. Since headlines are capitalized different from regular text, you may want to provide an `SGML` file that contains information about headline. This file uses the NIST format, and may be identical to source test sets provided by the NIST or other evluation campaigns. A language `LANGUAGE` may also be specified, but only English (`en`) is currently supported.

2.4.13 Truecaser

Instead of lowercasing all training and test data, we may also want to keep words in their natural case, and only change the words at the beginning of their sentence to their most frequent form. This is what we mean by truecasing. Again, this requires first the training of a truecasing model, which is a list of words and the frequency of their different forms.

```
train-truecaser.perl --model MODEL --corpus CASED
```

The model is trained from the cased (but tokenized) training corpus `CASED` and stored in the file `MODEL`.

Input to the decoder has to be truecased with the command

```
truecase.perl --model MODEL < IN > OUT
```

Output from the decoder has to be restored into regular case. This simply uppercases words at the beginning of sentences:

```
detruecase.perl < in > out [--headline SGML]
```

An SGML file with headline information may be provided, as done with the recaser.

2.4.14 Searchgraph to DOT

This small tool converts Moses searchgraph (`-output-search-graph FILE` option) to dot format. The dot format can be rendered using the graphviz[8] tool dot.

```
moses ... --output-search-graph temp.graph -s 3
   # we suggest to use a very limited stack size, -s 3
sg2dot.perl [--organize-to-stacks] < temp.graph > temp.dot
dot -Tps temp.dot > temp.ps
```

Using `-organize-to-stacks` makes nodes in the same stack appear in the same column (this slows down the rendering, off by default).

Caution: the input must contain the searchgraph of one sentence only.

2.5 Advanced Features of the Decoder

The basic features of the decoder are explained in the Tutorial (page 15). Here, we describe some additional features that have been demonstrated to be beneficial in some cases.

2.5.1 Content

- Lexicalized Reordering Models (page 52)
- Binary Phrase Tables with On-demand Loading (page 54)
- Binary Reordering Tables with On-demand Loading (page 55)
- XML Markup (page 55)
- Generating n-Best Lists (page 57)
- Word-to-word alignment (page 58)
- Minimum Bayes Risk Decoding (page 59)
- Handling Unknown Words (page 60)
- Output Search Graph (page 60)
- Early Discarding of Hypotheses (page 62)
- Maintaining stack diversity (page 63)
- Cube Pruning (page 63)
- Specifying Reordering Constraints (page 64)
- Multiple Translation Tables (page 65)
- Pruning the Translation Table (page 66)
- Build Instructions (page 66)

[8] http://www.graphviz.org/

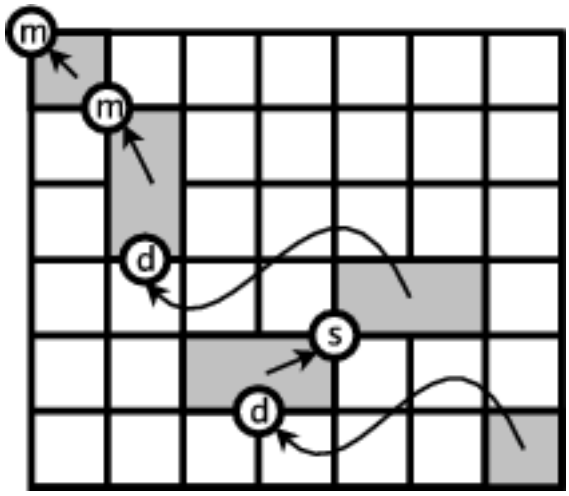
- Usage Instructions (page 66)
- Multi-threaded Moses (page 67)
- Moses Server (page 68)
- Amazon EC2 cloud (page 68)

2.5.2 Lexicalized Reordering Models

The default standard model that for phrase-based statistical machine translation is only conditioned on movement distance and nothing else. However, some phrases are reordered more frequently than others. A French adjective like *extérieur* typically gets switched with the preceding noun, when translated into English.

Hence, we want to consider a **lexicalized reordering model** that conditions reordering on the actual phrases. One concern, of course, is the problem of sparse data. A particular phrase pair may occur only a few times in the training data, making it hard to estimate reliable probability distributions from these statistics.

Therefore, in the lexicalized reordering model we present here, we only consider three reordering types: (m) monotone order, (s) switch with previous phrase, or (d) discontinuous. See below for an illustration of these three different types of **orientation** of a phrase.



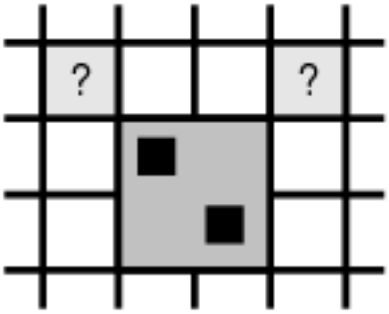
To put if more formally, we want to introduce a reordering model p_o that predicts an orientation type $\{m,s,d\}$ given the phrase pair currently used in translation:

$orientation \in \{m, s, d\}$

$p_o(orientation|f,e)$

How can we learn such a probability distribution from the data? Again, we go back to the word alignment that was the basis for our phrase table. When we extract each phrase pair, we can also extract its orientation type in that specific occurrence.

Looking at the word alignment matrix, we note for each extracted phrase pair its corresponding orientation type. The orientation type can be detected, if we check for a word alignment point to the top left or to the top right of the extracted phrase pair. An alignment point to the top left signifies that the preceding English word is aligned to the preceding Foreign word. An alignment point to the top right indicates that the preceding English word is aligned to the following french word. See below for an illustration.



The orientation type is defined as follows:

- **monotone**: if a word alignment point to the top left exists, we have evidence for monotone orientation.
- **swap**: if a word alignment point to the top right exists, we have evidence for a swap with the previous phrase.
- **discontinuous**: if neither a word alignment point to top left nor to the top right exists, we have neither monotone order nor a swap, and hence evidence for discontinuous orientation.

We count how often each extracted phrase pair is found with each of the three orientation types. The probability distribution p_o is then estimated based on these counts using the maximum likelihood principle:

p_o*(orientation|f,e) = count(orientation,e,f) /* Σ_o *count(o,e,f)*

Given the sparse statistics of the orientation types, we may want to smooth the counts with the unconditioned maximum-likelihood probability distribution with some factor Σ:

p_o*(orientation) =* Σ_f Σ_e *count(orientation,e,f) /* Σ_o Σ_f Σ_e *count(o,e,f)*

$p_o(\text{orientation}|f,e) = (\sigma p(\text{orientation}) + count(\text{orientation}, e, f))/(\sigma + \Sigma_o count(o, e, f))$

There are a number of variations of this lexicalized reordering model based on orientation types:

- **bidirectional**: Certain phrases may not only flag, if they themselves are moved out of order, but also if subsequent phrases are reordered. A lexicalized reordering model for this decision could be learned in addition, using the same method.

- **f** and **e**: Out of sparse data concerns, we may want to condition the probability distribution only on the foreign phrase (**f**) or the English phrase (**e**).
- **monotonicity**: To further reduce the complexity of the model, we might merge the orientation types swap and discontinuous, leaving a binary decision about the phrase order.

These variations have shown to be occasionally beneficial for certain training corpus sizes and language pairs. Moses allows the arbitrary combination of these decisions to define the reordering model type (e.g. `bidirectional-monotonicity-f`). See more on training these models in the training section of this manual.

2.5.3 Binary Phrase Tables with On-demand Loading

For larger tasks the phrase tables usually become huge, typically too large to fit into memory. Therefore, moses supports a binary phrase table with on-demand loading, i.e. only the part of the phrase table that is required to translate a sentence is loaded into memory.

You have to convert the standard ascii phrase tables into the binary format. Here is an example (standard phrase table `phrase-table`, with 5 scores):

```
cat phrase-table | sort | mosesdecoder/misc/processPhraseTable \
  -ttable 0 0 - -nscores 5 -out phrase-table
```

Options:

- `-ttable int int string` – translation table file, use `'-'` for stdin
- `-out string` – output file name prefix for binary ttable
- `-nscores int` – number of scores in ttable

If you just want to convert a phrase table, the two ints in the -ttable option do not matter, so use 0's.

Important: Make sure you set the environment variable `LC_ALL=C` for sorting. If your phrase table is already sorted, you can skip that.

The output files will be:

```
phrase-table.binphr.idx
phrase-table.binphr.srctree
phrase-table.binphr.srcvoc
phrase-table.binphr.tgtdata
phrase-table.binphr.tgtvoc
```

In the moses config file, specify `phrase-table` as phrase table. Moses will check if the binary version exists and use it.

Word-to-word alignment: To include in the binary phrase table the word-to-word alignments between source and target phrases which are contained in the textual phrase table (see Training Step "Score Phrases" (page 85)), specify the option `-alignment-info` in the processPhraseTable command. The two output files ".srctree" and " .tgtdata" will end with the suffix ".wa".

Note:

- if your textual PT does NOT contain w2w alignments, you CANNOT use "-alignment-info" to output w2w alignments in the binary format: you get an error message
- if your textual PT does contain w2w alignments you CAN use "-alignment-info" to create binary PT WITH w2w alignments: two of the binary PT data files (srctree and tgtdata) have suffix (.wa)
- if your textual PT does contain w2w alignments you CAN avoid "-alignment-info" to create binary PT WITHOUT w2w alignments
- by comparing data files of binary PT with and without w2w alignments, three of them (idx, srcvoc and tgtvoc) are identical and two of them (srctree and tgtdata) differ

Moses will check if the binary version with word-to-word alignments exists and optionally use it through the options `use-alignment-info`, `-print-alignment-info` and `-print-alignment-info-in-n-be`

2.5.4 Binary Reordering Tables with On-demand Loading

The reordering tables may be also converted into a binary format. The command is slightly simpler:

```
mosesdecoder/misc/processLexicalTable -in reordering-table -out reordering-table
```

The file names for input and output are typically the same, since the actual output file names have similar extensions to the phrase table file names.

2.5.5 XML Markup

Sometimes we have external knowledge that we want to bring to the decoder. For instance, we might have a better translation system for translating numbers of dates. We would like to plug in these translations to the decoder without changing the model.

The `-xml-input` flag is used to activate this feature. It can have one of four values:

- `exclusive` Only the XML-specified translation is used for the input phrase. Any phrases from the phrase table that overlap with that span are ignored.
- `inclusive` The XML-specified translation competes with all the phrase table choices for that span.

- `ignore` The XML-specified translation is ignored completely.
- `pass-through` (default) For backwards compatibility, the XML data is fed straight through to the decoder. This will produce erroneous results if the decoder is fed data that contains XML markup.

The decoder has an XML markup scheme that allows the specification of translations for parts of the sentence. In its simplest form, we can tell the decoder what to use to translate certain words or phrases in the sentence:

```
% echo 'das ist <np translation="a cute place">ein kleines haus</np>' \
  | moses -xml-input exclusive -f moses.ini
this is a cute place

% echo 'das ist ein kleines <n translation="dwelling">haus</n>' \
  | moses -xml-input exclusive -f moses.ini
this is a little dwelling
```

The words have to be surrounded by tags, such as <np...> and </np>. The name of the tags can be chosen freely. The target output is specified in the opening tag as a parameter value for a parameter that is called `english` for historical reasons (the canonical target language).

We can also provide a probability along with these translation choice. The parameter must be named `prob` and should contain a single float value. If not present, an XML translation option is given a probability of 1.

```
% echo 'das ist ein kleines <n translation="dwelling" prob="0.8">haus</n>' \
  | moses -xml-input exclusive -f moses.ini \
this is a little dwelling
```

This probability isn't very useful without letting the decoder have other phrase table entries "compete" with the XML entry, so we switch to `inclusive` mode. This allows the decoder to use either translations from the model or the specified xml translation:

```
% echo 'das ist ein kleines <n translation="dwelling" prob="0.8">haus</n>' \
  | moses -xml-input inclusive -f moses.ini
this is a small house
```

The switch `-xml-input inclusive` gives the decoder a choice between using the specified translations or its own. This choice, again, is ultimately made by the language model, which takes the sentence context into account.

This doesn't change the output from the non-XML sentence because that `prob` value is first logged, then split evenly among the number of scores present in the phrase table. Additionally, in the toy model used here, we are dealing with a very dumb language model and phrase table. Setting the probability value to something astronomical forces our option to be chosen.

```
% echo 'das ist ein kleines <n translation="dwelling" prob="0.8">haus</n>' \
  | moses -xml-input inclusive -f moses.ini
this is a little dwelling
```

The XML-input implementation is **NOT** currently compatible with factored models or confusion networks.

Options

- `-xml-input` ('pass-through' (default), 'inclusive', 'exclusive', 'ignore')

2.5.6 Generating n-Best Lists

The generation of n-best lists (the top n translations found by the search according to the model) is pretty straight-forward. You simple have to specify the file where the n-best list will be stored and the size of the n-best list for each sentence.

Example: The command

```
% moses -f moses.ini -n-bestlist listfile 100 < in
```

stores the n-best list in the file `listfile` with up to 100 translations per input sentence.

Here an example n-best list:

```
0 ||| we must discuss on greater vision .  ||| d: 0 -5.56438 0 0 -7.07376 0 0 \
  lm: -36.0974 -13.3428 tm: -39.6927 -47.8438 -15.4766 -20.5003 4.99948 w: -7 ||| -9.2298
0 ||| we must also discuss on a vision .  ||| d: -10 -2.3455 0 -1.92155 -3.21888 0 -1.51918 \
  lm: -31.5841 -9.96547 tm: -42.3438 -48.4311 -18.913 -20.0086 5.99938 w: -8 ||| -9.26197
0 ||| it is also discuss a vision .  ||| d: -10 -1.63574 -1.60944 -2.70802 -1.60944 -1.94589
  lm: -31.9699 -12.155 tm: -40.4555 -46.8605 -14.3549 -13.2247 4.99948 w: -7 ||| -9.31777
```

Each line of the n-best list file is made up of (separated by |||):

- sentence number (in above example 0, the first sentence)
- output sentence
- individual component scores (unweighted)
- weighted overall score

Note that it is possible (and very likely) that the n-best list contains many sentences that look the same on the surface, but have different scores. The most common reason for this is different phrase segmentation (two words may be mapped by a single phrase mapping, or by two individual phrase mappings for each word).

To produce an n-best list that only contains the first occurrence of an output sentence, add the word `distinct` after the file and size specification:

```
% moses -f moses.ini -n-bestlist listfile 100 distinct < in
```

This creates an n-best list file that contains up to 100 distinct output sentences for each input sentences. Note that potentially a large numbers of candidate translations have to be examined to find the top 100. To keep memory usage in check only 20 times the specified number of distinct entries are examined. This factor can be changed with the switch `-n-best-factor`.

Options

- `-n-best-list FILE SIZE [distinct]` — output an n-best list of size `SIZE` to file `FILE`
- `-n-best-factor FACTOR` — exploring at most `FACTOR*SIZE` candidates for distinct
- `-include-alignment-in-n-best` — output of word-to-word alignments in the n-best list; it requires that w2w alignments are included in the phrase tabel and that `-use-alignment-info` is set. (See here (page 58) for further details).

2.5.7 Word-to-word alignment

If the phrase table (binary or textual) includes word-to-word alignments between source and target phrases (see "Score Phrases" (page 85) and "Binary Phrase Table" (page 54)), Moses can report them in the output.

There are three options that control the output of alignment infotmation: `-use-alignment-info`, `-print-alignment-info`, and `-print-alignment-info-in-n-best`.

For instance, by translating the sentence `"ich frage"` from German into English and activating all parameters, you get in the verbose output:

```
BEST TRANSLATION: i ask [11]  [total=-1.429] <<features>> [f2e: 0=0 1=1] [e2f: 0=0 1=1]
```

The last two fields report the word-to-word alignments from source to target and from target to source, respectively.

In the n-best list you get:

```
0 ||| i ask  ||| ...feature_scores.... ||| -1.42906 ||| 0-1=0-1 ||| 0=0 1=1 ||| 0=0 1=1
0 ||| i am asking  ||| ...feature_scores.... ||| -2.61281 ||| 0-1=0-2 ||| 0=0 1=1,2 ||| 0=0
0 ||| i ask you  ||| ...feature_scores.... ||| -3.1068 ||| 0-1=0-2 ||| 0=0 1=1,2 ||| 0=0 1=
0 ||| i ask this  ||| ...feature_scores.... ||| -3.48919 ||| 0-1=0-2 ||| 0=0 1=1 ||| 0=0 1=
```

Indexes (starting from 0) are used to refer to words. '2=-1' means that the word of index 2 (i.e. the word) is not associated with any word in the other language. For instance, by considering the last translation hypothesis "i ask this" of "ich frage", the source to target alignment ("0=0 1=1") means that:

```
German   -> English
ich      -> i
frage    -> ask
```

and viceversa the target to source alignment ("0=0 1=1 2=-1") means that:

```
English  -> German
i        -> ich
ask      -> frage
this      ->
```

Note: in the same translation hypothesis, the the field "0-1=0-2" after the global score refers to the phrase-to-phrase alignment and means that "ich frage" is translated as a phrase into the three-word English phrase "i ask you". This information is generated if the option `-include-alignment-in-n-best` is activated.

Important: the phrase table can include different word-to-word alignments for the source-to-target and target-to-source directions, at least in principle. Hence, the two alignments can differ.

Options

- `-use-alignment-info` – to activate this feature
- `-print-alignment-info` – to output the word-to-word alignments into the verbose output.
- `-print-alignment-info-in-n-best` – to output the word-to-word alignments into the verbose output.

2.5.8 Minimum Bayes Risk Decoding

Minumum Bayes Risk (MBR) decoding was proposed by Shankar Kumar and Bill Byrne (HLT/NAACL 2004)[9]. Roughly speaking, instead of outputting the translation with the highest probability, MBR decoding outputs the translation that is most similar to the most likely translations. This requires a similarity measure to establish *similar*. In Moses, this is a smoothed BLEU score.

Using MBR decoding is straight-forward, just use the switch `-mbr` when invoking the decoder.

Example:

```
% moses -f moses.ini -mbr < in
```

[9] http://mi.eng.cam.ac.uk/ wjb31/ppubs/hlt04_mbr_smt.pdf

MBR decoding uses by default the top 200 distinct candidate translations to find the translation with minimum Bayes risk. If you want to change this to some other number, use the switch `-mbr-size`:

```
% moses -f moses.ini -decoder-type 1 -mbr-size 100 < in
```

MBR decoding requires that the translation scores are converted into probabilities that add up to one. The default is to take the log-scores at face value, but you may get better results with scaling the scores. This may be done with the switch `-mbr-scale`, so for instance:

```
% moses -f moses.ini -decoder-type 1 -mbr-scale 0.5 < in
```

Options

- `-mbr` – use MBR decoding
- `-mbr-size` SIZE – number of translation candidates consider (default 200)
- `-mbr-scale` SCALE – scaling factor used to adjust the translation scores (default 1.0)

2.5.9 Handling Unknown Words

Unknown words are copied verbatim to the output. They are also scored by the language model, and may be placed out of order. Alternatively, you may want to drop unknown words. To do so add the switch `-drop-unknown`.

When translating between languages that use different writing sentences (say, Chinese-English), dropping unknown words results in better BLEU scores. However, it is misleading to a human reader, and it is unclear what the effect on human judgment is.

Options

- `-drop-unknown` – drop unknown words instead of copying them into the output

2.5.10 Output Search Graph

It may be useful for many downstream applications to have a dump of the search graph, for instance to compile a word lattice. One the one hand you can use the `-verbose 3` option, which will give a trace of all generated hypotheses, but this creates logging of many hypotheses that get immediately discarded. If you do not want this, a better option is using the switch `-output-search-graph FILE`, which also provides some additional information.

The generated file contains lines that could be seen as both a dump of the states in the graph and the transitions in the graph. The state graph more closely reflects the hypotheses that are generated in the search. There are three types of hypotheses:

- The initial empty hypothesis is the only one that is not generated by a phrase translation

```
0 hyp=0 stack=0 [...]
```

- Regular hypotheses

```
0 hyp=17 stack=1 back=0 score=-1.33208 [...] covered=0-0 out=from now on
```

- Recombined hypotheses

```
0 hyp=5994 stack=2 back=108 score=-1.57388 [...] recombined=13061 [...] covered=2-2 out=be
```

The relevant information for viewing each line as a state in the search graph is the sentence number (initial `0`), the hypothesis id (`hyp`), the stack where the hypothesis is placed (same as number of foreign words covered, `stack`), the back-pointer to the previous hypotheses (`back`), the score so far (`score`), the last output phrase (`out`) and that phrase's foreign coverage (`covered`). For recombined hypotheses, also the superior hypothesis id is given (`recombined`).

The search graph output includes additional information that is computed after the fact. While the backpointer and score (`back`, `score`) point to the cheapest path and cost to the beginning of the graph, the generated output also inclused the pointer to the cheapest path and score (`forward`, `fscore`) to the end of the graph.

One way to view the output of this option is a reflection of the search and all (relevant) hypotheses that are generated along the way. But often, we want to generate a word lattice, where the states are less relevant, but the information is in the transitions from one state to the next, each transition emitting a phrase at a certain cost. The initial empty hypothesis is irrelevant here, so we need to consider only the other two hypothesis types:

- Regular hypotheses

```
0 hyp=17 [...] back=0 [...] transition=-1.33208 [...] covered=0-0 out=from now on
```

- Recombined hypotheses

```
0 [...] back=108 [...] transition=-0.640114 recombined=13061 [...] covered=2-2 out=be
```

For the word lattice, the relevant information is the cost of the transition (`transition`), its output (`out`), maybe the foreign coverage (`covered`), and the start (`back`) and endpoint (`hyp`). Note that the states generated by recombined hypothesis are ignored, since the transition points to the superior hypothesis (`recombined`).

Here, for completeness sake, the full lines for the three examples we used above:

```
0 hyp=0 stack=0 forward=9 fscore=-107.279
0 hyp=17 stack=1 back=0 score=-1.33208 transition=-1.33208 \
  forward=517 fscore=-106.484 covered=0-0 out=from now on
0 hyp=5994 stack=2 back=108 score=-1.57388 transition=-0.640114 \
  recombined=13061 forward=22455 fscore=-106.807 covered=2-2 out=be
```

What is the difference between the search graph output file generated with this switch and the true search graph?

- It contains the additional forward costs and forward paths.
- It also only contains the hypotheses that are part of a fully connected path from the initial empty hypothesis to a final hypothesis that covers the full foreign input sentence.
- The recombined hypotheses already point to the correct superior hypothesis, while the `-verbose 3` log shows the recombinations as they happen (recall that momentarily superior hypotheses may be recombined to even better ones down the road).

Note again that you can get the full search graph with the `-verbose 3` option. It is, however, much larger and mostly consists of discarded hypotheses.

Options

- `-output-search-graph FILE` – output the search graph for each sentence in a file

2.5.11 Early Discarding of Hypotheses

During the beam search, many hypotheses are created that are too bad to be even entered on a stack. For many of them, it is even clear before the construction of the hypothesis that it would be not useful. Early discarding of such hypotheses hazards a guess about their viability. This is based on correct score except for the actual language model costs which are very expensive to compute. Hypotheses that, according to this estimate, are worse than the worst hypothesis of the target stack, even given an additional specified threshold as cushion, are not constructed at all. This often speeds up decoding significantly. Try threshold factors between 0.5 and 1.

Options

- `-early-discarding-threshold THRESHOLD` – use early discarding of hypotheses with the specified threshold (default: 0 = not used)

2.5.12 Maintaining stack diversity

The beam search organizes and compares hypotheses based on the number of foreign words they have translated. Since they may have different foreign words translated, we use future score estimates about the remaining sentence translation score.

Instead of comparing such apples and oranges, we could also organize hypotheses by their exact foreign word coverage. The disadvantage of this is that it would require an exponential number of stacks, but with reordering limits the number of stacks is only exponential with regard to maximum reordering distance.

Such coverage stacks are implemented in the search, and their maximum size is specified with the switch `-stack-diversity` (or `-sd`), which sets the maximum number of hypotheses per coverage stack.

The actual implementation is a hybrid of coverage stacks and foreign word count stacks: the stack diversity is a constraint on which hypotheses are kept on the traditional stack. If the stack diversity limits leave room for additional hypotheses according to the stack size limit (specified by `-s`, default 200), then the stack is filled up with the best hypotheses, using score so far and the future score estimate.

Options

- `-stack-diversity LIMIT` – keep a specified number of hypotheses for each foreign word coverage (default: 0 = not used)

2.5.13 Cube Pruning

Cube pruning, as described by Liang Huang and David Chiang (2007), has been implemented in the Moses decoder. This is in addition to the traditional search algorithm. The code offers developers the opportunity to implement different search algorithms using an extensible framework.

Cube pruning is faster than the traditional search at comparable levels of search errors. To get faster performance than the default Moses setting at roughly the same performance, use the parameter settings:

```
-search-algorithm 1 -cube-pruning-pop-limit 2000 -s 2000
```

This uses cube pruning (`-search-algorithm`) that adds 2000 hypotheses to each stack (`-cube-pruning-pop-li 2000`) and also increases the stack size to 2000 (`-s 2000`). Note that with cube pruning, the size of the stack has little impact on performance, so it should be set rather high. The speed/quality trade-off is mostly regulated by the cube pruning pop limit, i.e. the number of hypotheses added to each stack.

Stacks are organized by the number of foreign words covered, so they may differ by which words are covered. You may also require that a minimum number of hypotheses is added for each word coverage (they may be still pruned out, however). This is done using the switch `-cube-pruning-diversity MINIMUM` which sets the minimum. The default is 0.

Options

- `-search-algorithm 1` – turns on cube pruning
- `-cube-pruning-pop-limit LIMIT` – number of hypotheses added to each stack
- `-cube-pruning-diversity MINIMUM` – minimum number of hypotheses from each coverage pattern

2.5.14 Specifying Reordering Constraints

For various reasons, it may be useful to specify reordering constraints to the decoder, for instance because of punctuation. Consider the sentence:

```
I said " This is a good idea . " , and pursued the plan .
```

The quoted material should be translated as a block, meaning that once we start translating some of the quoted words, we need to finish all of them. We call such a block a **zone** and allow the specification of such constraints using XML markup.

```
I said <zone> " This is a good idea . " </zone> , and pursued the plan .
```

Another type of constraints are **walls** which are hard reordering constraints: First all words before a wall have to be translated, before words afterwards are translated. For instance:

```
This is the first part . <wall /> This is the second part .
```

Walls may be specified within zones, where they act as **local walls**, i.e. they are only valid within the zone.

```
I said <zone> " <wall /> This is a good idea . <wall /> " </zone> , and pursued the plan .
```

If you add such markup to the input, you need to use the option `-xml-input` with either `exclusive` or `inclusive` (there is no difference between these options in this context).

Specifying reordering constraints around punctuation is often a good idea. The switch `-monotone-at-punctuation` introduces walls around the punctuation tokens `,.!?:;"`.

Options

- walls and zones have to specified in the input using the tags <zone>, </zone>, and <wall>.
- -xml-input – needs to be exclusive or inclusive
- -monotone-at-punctuation (-mp) – adds walls around punctuation ,.!?:;".

2.5.15 Multiple Translation Tables

Moses allows the use of multiple translation tables, but there are two different ways how they are used:

- **both** translation tables are used for scoring: This means that every translation option is collected from each table and scored by each table. This implies that each translation option has to be contained in each table: if it is missing in one of the tables, it can not be used.
- **either** translation table is used for scoring: Translation options are collected from one table, and additional options are collected from the other tables. If the same translation option (in terms of identical input phrase and output phrase) is found in multiple tables, separate translation options are created for each occurrence, but with different scores.

In any case, each translation table has its own set of weights.

First, you need to specify the translation tables in the section [ttable-file] of the moses.ini configuration file, for instance:

```
[ttable-file]
0 0 5 /my-dir/table1
0 0 5 /my-dir/table2
```

Secondly, you need to set the appropriate number of weights in the section [weight-t], in our example that would be 10 weights (5 for each table).

Thirdly, you need to specify how the tables are used in the section [mapping]. As mentioned above, there are two choices:

- scoring with **both** tables:

```
[mapping]
T 0
T 1
```

- scoring with **either** table:

```
[mapping]
0 T 0
1 T 1
```

Note: what we are really doing here is using Moses' capabilities to use different encoding paths. The number before "T" defines a decoding path, so in the second example are two different decoding paths specified. Decoding paths may also contain additional mapping steps, such as generation steps and translation steps using different factors.

Also note that there is no way to have the option "use both tables, if the phrase pair is in both table, otherwise use only the table where you can find it". Keep in mind, that scoring a phrase pair involves a cost and lowers the chances that the phrase pair is used. To effectively use this option, you may create a third table that consists of the intersection of the two phrase tables, and remove shared phrase pairs from each table.

2.5.16 Pruning the Translation Table

The translation table contains all phrase pairs found in the parallel corpus, which includes a lot of noise. To reduce the noise, recent work by Johnson et al. has suggested to prune out unlikely phrase pairs. For more detail, please refer to the paper:

H. Johnson, J. Martin, G. Foster and R. Kuhn. (2007) "'Improving Translation Quality by Discarding Most of the Phrasetable"'. In Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), pp. 967-975.

Build Instructions

Moses includes a re-implementation of this method in the directory `sigtest-filter`. You first need to build it from the source files.

This implementation relies on Joy Zhang's SALM Suffix Array toolkit[10].

1. download and extract the SALM source release.
2. in `SALM/Distribution/Linux` type: `make`
3. enter the directory `sigtest-filter` in the main Moses distribution directory
4. type make `SALMDIR=/path/to/SALM`

Usage Instructions

Using the `SALM/Bin/Linux/Index/IndexSA.O32`, create a suffix array index of the source and target sides of your training bitext (`SOURCE`, `TARGET`).

[10] http://projectile.sv.cs.cmu.edu/research/public/tools/salm/salm.htm

```
% SALM/Bin/Linux/Index/IndexSA.O32 TARGET
% SALM/Bin/Linux/Index/IndexSA.O32 SOURCE
```

Prune the phrase table:

```
% cat phrase-table | ./filter-pt -e TARGET -f SOURCE -l FILTER-VALUE > phrase-table.pruned
```

FILTER-VALUE is the -log prob threshold described in Johnson et al. (2007)'s paper. It may be either 'a+e', 'a-e', or a positive real value. Run with no options to see more use-cases. A good setting is -l a+e -n 30, which also keeps only the top 30 phrase translations for each source phrase, based on p(e|f).

2.5.17 Multi-threaded Moses

The latest svn version of moses now supports multi-threaded operation, enabling faster decoding on multi-core machines. The current limitations of mult-threaded moses are:

1. irstlm is not supported, since it uses a non-threadsafe cache
2. lattice input may not work - this has not been tested
3. increasing the verbosity of moses will probably cause mult-threaded moses to crash

To configure and build multi-threaded moses, you'll need to have boost[11] installed (1.35 or higher) and use the configure line

```
% ./configure --with-srilm=<path-to-srilm> --with-boost=<path-to-boost> --enable-threads
```

The boost path can be omitted if you have boost installed in a standard place. On 64-bit machines you may have to add -with-boost-thread=boost_thread-gcc43-mt (or similar) to the configure arguments.

After moses has been configured this way, running make will build two moses binaries, moses and mosesmt. The latter takes the same arguments as moses (although it doesn't currently support all of moses' i/o options) but it also admits an additional -threads n argument, specifying the size of the threadpool to use when running the decoder. Using a small number of threads (3-5) has been found to speed up decoding, although larger numbers do not seem to offer any further increase in speed. Multi-threaded moses is still experimental, and any feedback on its use would be greatly appreciated. Either mail me[12] or the moses list.

[11] http://www.boost.org
[12] mailto:bhaddow@inf.ed.ac.uk

2.5.18 Moses Server

The moses server enables you to run the decoder as a server process, and send it sentences to be translated via xmlrpc[13]. This means that one moses process can service distributed clients coded in Java, perl, python, php, or any of the many other languages which have xmlrpc libraries.

To build the moses server, you need to have xmlrpc-c[14] installed - it has been tested with the latest stable version, 1.16.19, and you need to add the argument `-with-xmlrpc-c=<path-xmlrpc-c-config>` to the configure arguments. You will also need to configure moses for mult-threaded operation, as described above.

Running make should then build an executable `server/mosesserver`. This can be launched using the same command-line arguments as moses, with two additional arguments to specify the listening port and log-file (`-server-port` and `-server-log`). These default to 8080 and `/dev/null` respectively.

A sample client is included in the `server` directory (in perl), which requires the `SOAP::Lite` perl module installed. To access the moses server, an xmlrpc request should be sent to http://host:port/RPC2 where the parameter is a map containing the keys `text` and (optionally) `align`. The value of the first of these parameters is the text to be translated and the second, if present, causes alignment information to be returned to the client. The client will receive a map containing the same two keys, where the value associated with the `text` key is the translated text, and the `align` key (if present) maps to a list of maps. The alignment gives the segmentation in target order, with each list element specifying the target start position (`tgt-start`), source start position (`src-start`) and source end position (`src-end`).

2.5.19 Amazon EC2 cloud

Achim Ruopp has created a package to run the Moses pipeline on the Amazon cloud. This would be very useful for people who don't have their own SGI cluster. More details from the Amazon webpage, or from Achim directly

`://developer.amazonwebservices.com/connect/entry.jspa?externalID=3058\&ca`

Achim has also created a tutorial

`://www.digitalsilkroad.net/walkthrough.pdf`

2.6 Translating Web pages with Moses

(Code and documentation written by Herve Saint-Amand.)

[13] http://www.xmlrpc.com/
[14] http://xmlrpc-c.sourceforge.net/

We describe a small set of publicly available Perl scripts that provide the mechanisms to translate a Web page by retrieving it, extracting all sentences it contains, stripping them of any font style markup, translating them using the Moses system, re-inserting them in the document while preserving the layout of the page, and presenting the result to the user, providing a seamless translation system comparable to those offered by Google, BabelFish and others.

2.6.1 Introduction

Purpose of this program

Moses is a cutting-edge machine translation program that reflects the latest developments in the area of statistical machine translation research. It can be trained to translate between any two languages, and yields high quality results. However, the Moses program taken alone can only translate plain text, i.e., text stripped of any formatting or style information (as in `.txt` files). Also, it only deals with a single sentence at a time.

A program that can translate Web pages is a very useful tool. However, Web pages contain a lot of formatting information, indicating the color, font and style of each piece of text, along with its position in the global layout of the page. Most Web pages also contain more than one sentence or independent segment of text. For these reasons a Web page cannot be fed directly to Moses in the hope of obtaining a translated copy.

The scripts described in this document implement a Web page translation system that, at its core, relies on Moses for the actual translation task. The scripts' job, given a Web page to translate, is to locate and extract all strings of text in the page, split paragraphs into individual sentences, remove and remember any style information associated with the text, send the normalized, plain-text string to Moses for translation, re-apply the style onto the translated text and re-insert the sentence at its place in the original page structure, (hopefully) resulting in a translation of the original.

A word of warning

These scripts are a proof-of-concept type of demonstration, and should not be taken for more than that. They most probably still contain bugs, and possibly even security holes. They are not appropriate for production environments.

Intended audience and system requirements

This document is meant for testers and system administrators who wish to install and use the scripts, and/or to understand how they work.

Before starting, the reader should ideally possess basic knowledge of:

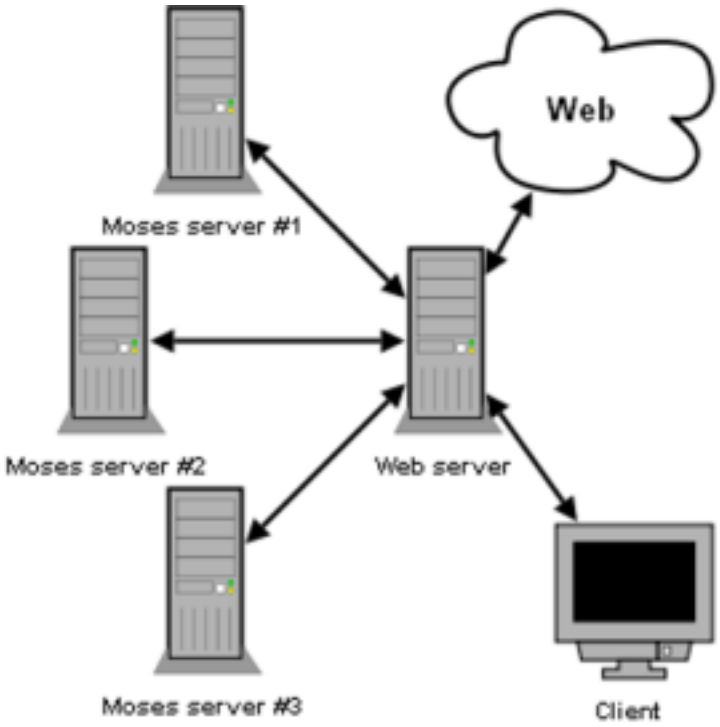
- UNIX-type command-line environments
- TCP/IP networking (know what a hostname and a port are)

- how to publish a Web page using a CGI script on an Apache server
- how to configure and operate the Moses decoder

and have the following resources available:

- an Apache (or similar) Web server
- the possibility of running CPU- and memory-intensive programs, either on the Web server itself (not recommended), or on one or several other machines that can be reached from the Web server
- Moses installed on those machines

Overview of the architecture



The following is a quick overview of how the whole system works. An attempt at illustrating the architecture is in the figure above. File names refer to files available from an SVN respository, as explained in the download section.

1. The Moses system is installed and configured on one or several computers that we designate as **Moses servers**.
2. On each Moses server, a daemon process, implemented by `daemon.pl`, accepts network connections on a given port and copies everything it gets from those connections straight

to Moses, sending back to the client what Moses printed back. This basically *plugs* Moses directly onto the network.

3. Another computer, which we designate as the **web server**, runs Apache (or similar) Web server software.

4. Through that server, the CGI scripts discussed in this document (`index.cgi`, `translate.cgi` and supporting files) are served to the client, providing the user interface to the system. It is a simple matter to configure `translate.cgi` so that it knows where the Moses servers are located.

5. A client requests `index.cgi` via the Web server. A form containing a textbox is served back, where the user can enter a URL to translate.

6. That form is submitted to `translate.cgi`, which does the bulk of the job. It fetches the page from the Web, extracts translatable plain text strings from it, sends those to the Moses servers for translation, inserts the translations back into the document, and serves the document back to the client. It adjusts links so that if any one is clicked in the translated document, a translated version will be fetched rather than the document itself.

The script containing all the interesting code, `translate.cgi`, is heavily commented, and programmers might be interested in reading it.

Mailing list

Should you encounter problems you can't solve during the installation and operation of this program, you can write to the moses support mailing list at `moses-support@mit.edu`. Should you not encounter problems, the author (whose email is found in the source file headers) would be astonished to hear about it.

2.6.2 Detailed setup instructions

Obtaining a copy of the scripts

The scripts are stored in an SVN repository. A copy can be retrieved by issuing the following command:

```
svn co://mosesdecoder.svn.sf.net/svnroot/mosesdecoder/trunk/web moses-web
```

This will place a copy of the source code in a new directory named `moses-web`. The whole source code consists in less than 100kB.

Setting up the Web server

The extracted source code is ready to be run, there is no installation procedure that compiles or copies files. The program is entirely contained within the directory that was downloaded

from SVN. It now needs to be placed on a Web server, in a properly configured location such that the CGI scripts (the two files bearing the `.cgi` extension) are executed when requested from a browser.

For instance, if you are on a shared Web server (e.g., a server provided by your university) and your user directory contains a directory named `public_html`, placing the `moses-web` directory inside `public_html` should make it available via the Web, at an address similar to http://www.dept.uni/ you/moses-web/.

Troubleshooting

- **404 Not Found** Perhaps the source code folder is not in the right location? Double-check the directory names. See if the home folder (parent of `moses-web` itself) is reachable. Ask your administrator.
- **403 Forbidden**, or you see the Perl source code of the script in your browser} The server is not configured to execute CGI scripts in this directory. Move `moses-web` to the `cgi-bin` subdirectory of your Web home, if it exists. Create a `.htacess` file in which you enable the `ExecCGI` option (see the Apache documentation).
- **Internal server error** Perhaps the scripts do not have the right permissions to be executed. Go in `moses-web` and type the command `chmod 755 *cgi`.

The scripts are properly installed once you can point your browser at the correct URL and you see the textbox in which you should enter the URL, and the 'Translate' button. Pressing the button won't work yet, however, as the Moses servers need to be installed and configured first.

Setting up the Moses servers

You now need to install Moses and the `daemon.pl` script on at least one machine.

Choosing machines for the Moses servers

Running Moses is a slow and expensive process, at least when compared to the world of Web servers where everything needs to be lightweight, fast and responsive. The machine selected for running the translator should have a recent, fast processor, and as many GBs of memory as possible (see the Moses documentation for more details).

Technically, the translations could be computed on the same machine that runs the Web server. However, the loads that Moses places on a system would risk seriously impacting the performance of the Web server. For that reason, we advise not running Moses on the same computer as the Web server, especially not if the server is a shared server, where several users host their files (such as Web servers typically provided by universities). In case of doubt we recommend you ask your local administrator.

For the sake of responsiveness, you may choose to run Moses on several machines at once. The burden of translation will then be split equally among all the hosts, thus more or less diving

the total translation time by the number of hosts used. If you have several powerful computers at your disposal, simply repeat the installation instructions that follow on each of the machines independently.

The main translation script, which runs on the Web server, will want to connect to the Moses servers via TCP/IP sockets. For this reason, the Moses servers must be reachable from the Web server, either directly or via SSH tunnels of other proxy mechanisms. Ultimately the translation script on the Web server must have a hostname/port address it can connect to for each Moses server.

Installing the scripts

Install Moses

For each Moses server, you will need to install and configure Moses for the language pair that you wish to use. If your Moses servers are all identical in terms of hardware, OS and available libraries, installing and training Moses on one machine and then copying the files over to the other ones should work, but your mileage may vary.

Install daemon.pl

Once Moses is working, check out, on each Moses server, another copy of the `moses-web` source directory by following again the instructions in the download section. Open `bin/daemon.pl`, and edit the `$MOSES` and `$MOSES_INI` paths to point to the location of your `moses` binary and your `moses.ini` configuration file.

Choose a port number

Now you must choose a port number for the daemon process to listen on. Pick any number between 1,024 and 49,151, ideally not a standard port for common programs and protocols to prevent interference with other programs (i.e., pick a port not mentioned in your `/etc/services` file).

Start the daemon

To activate a Moses server, simply type, in a shell running on that server:

```
./daemon.pl <hostname> <port>
```

where <`hostname`> is the name of the host you're typing this on (found by issuing the `hostname` command), and <`port`> is the port you selected. It may be misleading that despite its name, this program does not fork a background process, it is the background process itself. To truly launch the process in the background so that it continues running after the shell is closed, this command might be more useful:

```
nohup ./daemon.pl <hostname> <port> \&
```

The `bin/start-daemon-cluster.pl` script distributed with this program provides an automation mechanism that worked well in the original setup on the University of Saarland net-

work. It was used to start and stop the Moses servers all at once, also setting up SSH tunnelling on startup. Because it is very simple and trimmed to the requirements of that particular installation, we do not explain its use further here, but the reader might find inspiration in reading it.

Test the Moses servers

The daemon should now be listening on the port you chose. When it recieves a connection, it will read the input from that connection one line at a time, passing each line in turn to Moses for translation, and printing back the translation followed by a newline.

If you have the NetCat tool installed, you can test whether it worked by going to a shell on the Web server and typing `echo "Hello world" | nc <hostname> <port>`, replacing `Hello world` by a phrase in your source language if it is not English, and `<hostname>` and `<port>` by the values pointing to the Moses server you just set up. A translation should be printed back.

Configure the tokenizer

The `translate.cgi` script uses external tokenizer and detokenizer scripts. These scripts adapt their regexes depending on the language parsed, and so tokenizing is improved if the correct language is selected. This is done by opening `translate.cgi` with your favourite text editor, and setting `$INPUT_LANG` and `$OUTPUT_LANG` to the appropriate language codes. Currently the existing language codes are the file extensions found in the `bin/nonbreaking_prefixes` directory. If yours are not there, simply use `en` – end-of-sentence detection may then be suboptimal, and translation quality may be impacted, but the system will otherwise still function.

Configure the Web server to connect to the Moses servers

The last remaining step is to tell the frontend Web server where to find the backend Moses servers. Still in `translate.cgi`, set the `@MOSES_ADDRESSES` array to the list of `hostname:port` strings identifying the Moses servers.

Here is a sample valid configuration for three Moses servers named `server01`, `server02` and `server03`, each with the daemon listening on port 7070:

```
my @MOSES_ADDRESSES = ("server01:7070", "server02:7070", "server03:7070");
```

Stopping the daemons once done

The daemon processes continuously keep a copy of Moses running, so they consume memory even when idle. For this reason, we recommend that you stop them once they are not needed anymore, for instance by issuing this command on each Moses server: `killall daemon.pl`

3

Training Manual

3.1 Training

3.1.1 Content

- Training process (page 75)
- Running the training script (page 76)

3.1.2 Training process

We will start with an overview of the training process. This should give a feel for what is going on and what files are produced. In the following, we will go into more details of the options of the training process and additional tools.

The training process takes place in nine steps, all of them executed by the script

```
train-factored-phrase-model.perl
```

The nine steps are

1. Prepare data (45 minutes)
2. Run GIZA++ (16 hours)
3. Align words (2:30 hours)
4. Get lexical translation table (30 minutes)
5. Extract phrases (10 minutes)
6. Score phrases (1:15 hours)
7. Build lexicalized reordering model (1 hour)
8. Build generation models

9. Create configuration file (1 second)

If you are running on a machine with multiple processors, some of these steps can be considerably sped up with the following option:

```
--parallel
```

The run times mentioned in the steps refer to a recent training run on the 751'000 sentence, 16 million word German-English Europarl corpus, on a 3GHz Linux machine.

If you wish to experiment with translation in both directions, step 1 and 2 can be reused, starting from step 3 the contents of the model directory get direction-dependent. In other words run steps 1 and 2, then make a copy of the whole experiment directory and continue two trainings from step 3.

3.1.3 Running the training script

For an standard phrase model, you will typically run the training script as follows.

Run the training script:

```
train-factored-phrase-model.perl -root-dir . --corpus corpus/euro --f de --e en
```

There should be two files in the *corpus/* directory called *euro.de* and *euro.en*. These files should be sentence-aligned halfs of the parallel corpus. *euro.de* should contain the German sentences, and *euro.en* should contain the corresponding English sentences.

More on the training parameters (page 151) at the end of this manual. For corpus preparation, see the section on how to prepare training data (page 76).

3.2 Preparing Training Data

Training data has to be provided sentence aligned (one sentence per line), in two files, one for the foreign sentences, one for the English sentences:

```
>head -3 corpus/euro.*
==> corpus/euro.de <==
wiederaufnahme der sitzungsperiode
ich erklaere die am donnerstag , den 28. maerz 1996 unterbrochene
sitzungsperiode des europaeischen parlaments fuer wiederaufgenommen .
begruessung

==> corpus/euro.en <==
resumption of the session
i declare resumed the session of the european parliament adjourned
on thursday , 28 march 1996 .
welcome
```

A few other points have to be taken care of:

- unix commands require the environment variable `LC_ALL=C`
- one sentence per line, no empty lines
- sentences longer than 100 words (and their corresponding translations) have to be eliminated (note that a shorter sentence length limit will speed up training
- everything lowercased (use `lowercase.perl`)

3.2.1 Training data for factored models

You will have to provide training data in the format

```
word0factor0|word0factor1|word0factor2 word1factor0|word1factor1|word1factor2 ...
```

instead of the un-factored

```
word0 word1 word2
```

3.2.2 Cleaning the corpus

The script `clean-corpus-n.perl` is small script that cleans up a parallel corpus, so it works well with the training script.

It performs the following steps:

- removes empty lines
- removes redundant space characters
- drops lines (and their corresponding lines), that are empty, too short, too long or violate the 9-1 sentence ratio limit of GIZA++

The command syntax is:

```
clean-corpus-n.perl CORPUS L1 L2 OUT MIN MAX
```

For example: `clean-corpus-n.perl raw de en clean 1 50` takes the corpus files `raw.de` and `raw.en`, deletes lines longer than 50, and creates the output files `clean.de` and `clean.en`.

3.3 Factored Training

For training a factored model, you will specify a number of additional training parameters:

```
--alignment-factors FACTORMAP
--translation-factors FACTORMAPSET
--reordering-factors FACTORMAPSET
--generation-factors FACTORMAPSET
--decoding-steps LIST
```

Alignment factors

It is usually better to carry out the word alignment (step 2-3 of the training process) on more general word representations with rich statistics. Even successful word alignment with words stemmed to 4 characters have been reported. For factored models, this suggests that word alignment should be done only on either the surface form or the stem/lemma.

Which factors are used during word alignment is set with the `-alignment-factors` switch. Let us formally define the parameter syntax:

- FACTOR = [`0` - `9`]+
- FACTORLIST = FACTOR [, FACTOR]*
- FACTORMAP = FACTORLIST - FACTORLIST

The switch requires a FACTORMAP as argument, for instance `0-0` (using only factor 0 from source and target language) or `0,1,2-0,1` (using factors 0, 1, and 2 from the source language and 0 and 1 from the target language).

Typically you may want to train the word alignment using surface forms or lemmas.

3.3.1 Translation factors

Purpose of training factored translation model training is to create one or more translation tables between a subset of the factors. All translation tables are trained from the same word alignment, and are specified with the switch `-translation-factors`.

To define the syntax, we have to extend our parameter syntax with

- FACTORMAPSET = FACTORMAP[+FACTORMAP]*

since we want to specify muliple mappings.

One example is `-translation-factors 0-0+1-1,2`, which create the two tables

```
phrase-table.0-0.gz
phrase-table.1-1,2.gz
```

3.3.2 Reordering factors

Reordering tables can be trained with `-reordering-factors`. Syntax is the same as for translation factors.

3.3.3 Generation factors

Finally, we also want to create generation tables between target factors. Which tables to generate is specified with `-generation-factors`, which takes a FACTORMAPSET as a parameter. Note that this time the mapping is between target factors, not between source and target factors.

One example is `-generation-factors 0-1` with creates a generation table between factor factor 0 and 1.

3.3.4 Decoding steps

The mapping from source words in factored representation into target words in factored representation takes place in a number of mapping steps (either using a translation table or a generation table). These steps are specified with the switch `-decoding-steps LIST`.

For example `-decoding-steps t0,g0,t1,t2,g1` specifies that mapping takes place in form of an initial translation step using translation table 0, then a generation step using generation table 0, followed by two translation steps using translation tables 1 and 2, and finally a generation step using generation table 1. (The specific names `t0, t1, ...` are automatically assigned to translation tables in the order you define them with `-translation-factors`, and likewise for `g0` etc.)

3.4 Training Step 1: Prepare Data

The parallel corpus has to be converted into a format that is suitable to the GIZA++ toolkit. Two vocabulary files are generated and the parallel corpus is converted into a numberized format.

The vocabulary files contain words, integer word identifiers and word count information:

```
==> corpus/de.vcb <==
1       UNK     0
2       ,       928579
3       .       723187
4       die     581109
5       der     491791
6       und     337166
7       in      230047
8       zu      176868
9       den     168228
```

```
10      ich     162745

==> corpus/en.vcb <==
1       UNK     0
2       the     1085527
3       .       714984
4       ,       659491
5       of      488315
6       to      481484
7       and     352900
8       in      330156
9       is      278405
10      that    262619
```

The sentence-aligned corpus now looks like this:

```
> head -9 corpus/en-de-int-train.snt
1
3469 5 2049
4107 5 2 1399
1
10 3214 4 116 2007 2 9 5254 1151 985 6447 2049 21 44 141 14 2580 3
14 2213 1866 2 1399 5 2 29 46 3256 18 1969 4 2363 1239 1111 3
1
7179
306
```

A sentence pair now consists of three lines: First the frequency of this sentence. In our training process this is always 1. This number can be used for weighting different parts of the training corpus differently. The two lines below contain word ids of the foreign and the English sentence. In the sequence `4107 5 2 1399` we can recognize `of (5)` and `the (2)`.

GIZA++ also requires words to be placed into word classes. This is done automatically by calling the `mkcls` program. Word classes are only used for the IBM reordering model in GIZA++. A peek into the foreign word class file:

```
> head corpus/de.vcb.classes
!       14
"       14
#       30
%       31
\&       10
'       14
(       10
)       14
+       31
,       11
```

3.5 Training Step 2: Run GIZA++

GIZA++ is a freely available implementation of the IBM Models. We need it as a initial step to establish word alignments. Our word alignments are taken from the intersection of bidirectional runs of GIZA++ plus some additional alignment points from the union of the two runs.

Running GIZA++ is the most time consuming step in the training process. It also requires a lot of memory (1-2 GB RAM is common for large parallel corpora).

GIZA++ learns the translation tables of IBM Model 4, but we are only interested in the word alignment file:

```
> zcat giza.de-en/de-en.A3.final.gz | head -9
# Sentence pair (1) source length 4 target length 3 alignment score : 0.00643931
wiederaufnahme der sitzungsperiode
NULL ({ }) resumption ({ 1 }) of ({ }) the ({ 2 }) session ({ 3 })
# Sentence pair (2) source length 17 target length 18 alignment score : 1.74092e-26
ich erklaere die am donnerstag , den 28. maerz 1996 unterbrochene sitzungsperiode
  des europaeischen parlaments fuer wiederaufgenommen .
NULL ({ 7 }) i ({ 1 }) declare ({ 2 }) resumed ({ }) the ({ 3 }) session ({ 12 })
  of ({ 13 }) the ({ }) european ({ 14 }) parliament ({ 15 })
  adjourned ({ 11 16 17 }) on ({ }) thursday ({ 4 5 }) , ({ 6 }) 28 ({ 8 })
  march ({ 9 }) 1996 ({ 10 }) . ({ 18 })
# Sentence pair (3) source length 1 target length 1 alignment score : 0.012128
begruessung
NULL ({ }) welcome ({ 1 })
```

In this file, after some statistical information and the foreign sentence, the English sentence is listed word by word, with references to aligned foreign words: The first word `resumption` `({ 1 })` is aligned to the first German word `wiederaufnahme`. The second word `of ({ })` is unaligned. And so on.

Note that each English word may be aligned to multiple foreign words, but each foreign word may only be aligned to at most one English word. This one-to-many restriction is reversed in the inverse GIZA++ training run:

```
> zcat giza.en-de/en-de.A3.final.gz | head -9
# Sentence pair (1) source length 3 target length 4 alignment score : 0.000985823
resumption of the session
NULL ({ }) wiederaufnahme ({ 1 2 }) der ({ 3 }) sitzungsperiode ({ 4 })
# Sentence pair (2) source length 18 target length 17 alignment score : 6.04498e-19
i declare resumed the session of the european parliament adjourned on thursday ,
  28 march 1996 .
NULL ({ }) ich ({ 1 }) erklaere ({ 2 10 }) die ({ 4 }) am ({ 11 })
  donnerstag ({ 12 }) , ({ 13 }) den ({ }) 28. ({ 14 }) maerz ({ 15 })
  1996 ({ 16 }) unterbrochene ({ 3 }) sitzungsperiode ({ 5 }) des ({ 6 7 })
```

```
  europaeischen ({ 8 }) parlaments ({ 9 }) fuer ({ }) wiederaufgenommen ({ })
  . ({ 17 })
# Sentence pair (3) source length 1 target length 1 alignment score : 0.706027
welcome
NULL ({ }) begruessung ({ 1 })
```

3.5.1 Training on really large corpora

GIZA++ is not only the slowest part of the training, it is also the most critical in terms of memory requirements. To better be able to deal with the memory requirements, it is possible to train a preparation step on parts of the data that involves an additional program called `snt2cooc`.

For practical purposes, all you need to know is that the switch `-parts n` may allow training on large corpora that would not be feasible otherwise (a typical value for `n` is 3).

This is currently not a problem for Europarl training, but is necessary for large Arabic and Chinese training runs.

3.5.2 Training in parallel

Using the `-parallel` option will fork the script and run the two directions of GIZA as independent processes. This is the best choice on a multi-processor machine.

If you have only single-processor machines and still wish to run the two GIZAs in parallel, use the following (rather obsolete) trick. Support for this is not fully userfriendly, some manual involvement is essential.

- First you start training the usual way with the additional switches `-last-step 2 -direction 1`, which runs the data perparation and one direction of GIZA++ training
- When the GIZA++ step started, start a second training run with the switches `-first-step 2 -direction 2`. This runs the second GIZA++ run in parallel, and then continues the rest of the model training. (Beware of race conditions! The second GIZA might finish earlier than the first one to training step 3 might start too early!)

3.6 Training Step 3: Align Words

To establish word alignments based on the two GIZA++ alignments, a number of heuristics may be applied. The default heuristic `grow-diag-final` starts with the intersection of the two alignments and then adds additional alignment points.

Other possible alignment methods:

- `intersection`

- `grow` (only add block-neighboring points)
- `grow-diag` (without final step)
- `union`
- `srctotgt` (only consider word-to-word alignments from the source-target GIZA++ alignment file)
- `tgttosrc` (only consider word-to-word alignments from the target-source GIZA++ alignment file)

Alternative alignment methods can be specified with the switch `-alignment`.

Here, the pseudo code for the default heuristic:

```
GROW-DIAG-FINAL(e2f,f2e):
 neighboring = ((-1,0),(0,-1),(1,0),(0,1),(-1,-1),(-1,1),(1,-1),(1,1))
 alignment = intersect(e2f,f2e);
 GROW-DIAG(); FINAL(e2f); FINAL(f2e);

GROW-DIAG():
 iterate until no new points added
   for english word e = 0 ... en
     for foreign word f = 0 ... fn
       if ( e aligned with f )
         for each neighboring point ( e-new, f-new ):
           if ( ( e-new not aligned or f-new not aligned ) and
                ( e-new, f-new ) in union( e2f, f2e ) )
             add alignment point ( e-new, f-new )
FINAL(a):
 for english word e-new = 0 ... en
   for foreign word f-new = 0 ... fn
     if ( ( e-new not aligned or f-new not aligned ) and
          ( e-new, f-new ) in alignment a )
       add alignment point ( e-new, f-new )
```

To illustrate this heuristic, see the example in the Figure below We with the intersection of the two alignments for the second sentence in the corpus above (left side) and then add some additional alignment points that lie in the union of the two alignments (right side).

```
#----------------- i          #----------------- i
-#---------------- declare    -#---------------- declare
------------------ resumed    ----------#------- resumed
--#--------------- the        --#--------------- the
-----------#------ session    -----------#------ session
------------#----- of         ------------#----- of
------------------ the        ------------#----- the
```

```
-------------#---- european   -------------#---- european
--------------#--- parliament --------------#--- parliament
------------------ adjourned  ---------------##- adjourned
------------------ on         ---#-------------- on
----#------------- thursday   ---##------------- thursday
-----#------------ ,          -----#------------ ,
-------#---------- 28         -------#---------- 28
--------#--------- march      --------#--------- march
---------#-------- 1996       ---------#-------- 1996
-----------------# .          -----------------# .

iedad,d2m1usdepfw.           iedad,d2m1usdepfw.
crimo e8a9nieuaui            crimo e8a9nieuaui
hke n n.r9ttsrrre            hke n n.r9ttsrrre
 l  n   z6ez ol d             l  n   z6ez ol d
 a  e     ru pa e             a  e     ru pa e
 r  r     bn am r             r  r     bn am r
 e  s     rg ie a             e  s     rg ie a
    t     os sn u                t     os sn u
```

This alignment has a blatant error: the alignment of the two verbs is mixed up. `resumed` is aligned to `unterbrochene`, and `adjourned` is aligned to `wiederaufgenommen`, but it should be the other way around.

To conclude this section, a quick look into the files generated by the word alignment process:

```
==> model/aligned.de <==
wiederaufnahme der sitzungsperiode
ich erklaere die am donnerstag , den 28. maerz 1996 unterbrochene sitzungsperiode
  des europaeischen parlaments fuer wiederaufgenommen .
begruessung

==> model/aligned.en <==
resumption of the session
i declare resumed the session of the european parliament adjourned on
  thursday , 28 march 1996 .
welcome

==> model/aligned.grow-diag-final <==
0-0 0-1 1-2 2-3
0-0 1-1 2-3 3-10 3-11 4-11 5-12 7-13 8-14 9-15 10-2 11-4 12-5 12-6 13-7
  14-8 15-9 16-9 17-16
0-0
```

The third file contains alignment information, one alignment point at a time, in form of the position of the foreign and English word.

3.7 Training Step 4: Get Lexical Translation Table

Given this alignment, it is quite straight-forward to estimate a maximum likelihood lexical translation table. We estimate the *w(e|f)* as well as the inverse *w(f|e)* word translation table. Here are the top translations for `europa` into English:

```
> grep ' europa ' model/lex.f2n | sort -nrk 3 | head
europe europa 0.8874152
european europa 0.0542998
union europa 0.0047325
it europa 0.0039230
we europa 0.0021795
eu europa 0.0019304
europeans europa 0.0016190
euro-mediterranean europa 0.0011209
europa europa 0.0010586
continent europa 0.0008718
```

3.8 Training Step 5: Extract Phrases

In the phrase extraction step, all phrases are dumped into one big file. Here is the top of that file:

```
> head model/extract
wiederaufnahme ||| resumption ||| 0-0
wiederaufnahme der ||| resumption of the ||| 0-0 1-1 1-2
wiederaufnahme der sitzungsperiode ||| resumption of the session ||| 0-0 1-1 1-2 2-3
der ||| of the ||| 0-0 0-1
der sitzungsperiode ||| of the session ||| 0-0 0-1 1-2
sitzungsperiode ||| session ||| 0-0
ich ||| i ||| 0-0
ich erklaere ||| i declare ||| 0-0 1-1
erklaere ||| declare ||| 0-0
sitzungsperiode ||| session ||| 0-0
```

The content of this file is for each line: foreign phrase, English phrase, and alignment points. Alignment points are pairs (foreign,english). Also, an inverted alignment file `extract.inv` is generated, and if the lexicalized reordering model is trained (default), a reordering file `extract.o`.

3.9 Training Step 6: Score Phrases

Subsequently, a translation table is created from the stored phrase translation pairs. The two steps are separated, because for larger translation models, the phrase translation table does not

fit into memory. Fortunately, we never have to store the phrase translation table into memory — we can construct it on disk.

To estimate the phrase translation probability $\phi(e|f)$ we proceed as follows: First, the extract file is sorted. This ensures that all English phrase translations for an foreign phrase are next to each other in the file. Thus, we can process the file, one foreign phrase at a time, collect counts and compute $\phi(e|f)$ for that foreign phrase *f*. To estimate $\phi(f|e)$, the inverted file is sorted, and then $\phi(f|e)$ is estimated for an English phrase at a time.

Next to phrase translation probability distributions $\phi(f|e)$ and $\phi(e|f)$, additional phrase translation scoring functions can be computed, e.g. lexical weighting, word penalty, phrase penalty, etc. Currently, lexical weighting is added for both directions and a fifth score is the phrase penalty.

```
> grep '| in europe |' model/phrase-table | sort -nrk 7 -t\| | head
in europa ||| in europe ||| 0.829007 0.207955 0.801493 0.492402 2.718
europas ||| in europe ||| 0.0251019 0.066211 0.0342506 0.0079563 2.718
in der europaeischen union ||| in europe ||| 0.018451 0.00100126 0.0319584 0.0196869 2.718
in europa , ||| in europe ||| 0.011371 0.207955 0.207843 0.492402 2.718
europaeischen ||| in europe ||| 0.00686548 0.0754338 0.000863791 0.046128 2.718
im europaeischen ||| in europe ||| 0.00579275 0.00914601 0.0241287 0.0162482 2.718
fuer europa ||| in europe ||| 0.00493456 0.0132369 0.0372168 0.0511473 2.718
in europa zu ||| in europe ||| 0.00429092 0.207955 0.714286 0.492402 2.718
an europa ||| in europe ||| 0.00386183 0.0114416 0.352941 0.118441 2.718
der europaeischen ||| in europe ||| 0.00343274 0.00141532 0.00099583 0.000512159 2.718
```

Currently, five different phrase translation scores are computed:

- phrase translation probability $\phi(f|e)$
- lexical weighting *lex(f|e)*
- phrase translation probability $\phi(e|f)$
- lexical weighting *lex(e|f)*
- phrase penalty (always *exp(1) = 2.718*)

Word-to-word alignment

An enhanced version of the scoring script outputs the word-to-word alignments between *f* and *e* as they are in the files (`extract` and `extract.inv`) generated in the previous training step "Extract Phrases" (page 85).

These two directional alignments are reported in the third and fourth fields; scores in the fifth field. In the third field, each word of the source phrase *f* is associated with the word(s) of the target phrase *e*, or with nothing. Vice versa, in the fourth field.

```
> grep '| in europe |' model/phrase-table | sort -nrk 7 -t\| | head
in europa ||| in europe ||| (0) (1) ||| (0) (1) ||| 0.829007 0.207955 ...
europas ||| in europe ||| (0,1) ||| (0) (0) ||| ...
in der europaeischen union ||| in europe ||| (0) () (1) (1) ||| (0) (2,3) ||| ..
in europa , ||| in europe ||| (0) (1) () ||| (0) (1) ||| ...
europaeischen ||| in europe ||| (1) ||| (0) () ||| ...
im europaeischen ||| in europe ||| (0) (1) ||| (0) (1) ||| ...
```

For instance:

```
in der europaeischen union ||| in europe ||| (0) () (1) (1) ||| (0) (2,3) ||| ...
```

means

```
German        -> English
in            -> in
der           ->
europaeischen -> europe
union         -> europe
```

and

```
English -> German
in      -> in
europe  -> europaeischen union
```

As the two word-to-word alignments come from one word alignment (see training step "Align words" (page 82)), the two fields represent the same information. However, they are independent in principle. Hence, you can change them at your pleasure; for example you could replace them with two different word-to-word (source-to-target and target-to-source).

3.10 Training Step 7: Build reordering model

By default, only a distance-based reordering model is included in final configuration. This model gives a cost linear to the reordering distance. For instance, skipping over two words costs twice as much as skipping over one word.

However, additional conditional reordering models may be build. These are conditioned on specified factors (in the source and target language), and learn different reordering probabilities for each phrase pair (or just the foreign phrase). Possible configurations are

- `msd` vs. `monotonicity`. MSD models consider three different orientation types: `monotone`, `swap`, and `discontinous`. Monotonicity models consider only `monotone` or `non-monotone`, in other words `swap`, and `discontinous` are lumped together.

- `f` vs. `fe`. The model may be conditioned on the foreign phrase (`f`), or on both the foreign phrase and English phrase (`fe`).
- `unidirectional` vs. `bidirectional`. For each phrase, the ordering of itself in respect to the previous is considered. For bidirectional models, also the ordering of the next phrase in respect to the currect phrase is modeled.

This gives us the following possible configuations:

- `msd-bidirectional-fe` (default)
- `msd-bidirectional-e`
- `msd-fe`
- `msd-f`
- `monotonicity-bidirectional-fe`
- `monotonicity-bidirectional-f`
- `monotonicity-fe`
- `monotonicity-f`

and of course `distance`.

Which reordering model is used (and built during the training process, if necessary) can be set with the switch `-reordering`, e.g.:

```
-reordering distance
-reordering distance,msd-bidirectional-fe
```

Note that the distance model is always included, so there is no need to specify it.

The number of features that are created with a lexical reordering model depends on the type of the model. A `msd` model has three features, one each for the probability that the phrase is translated monotone, swapped, or discontinuous. A `monotonicity` model has only one feature. If a `bidirectional` model is used, then the number of features doubles - one for each direction.

3.11 Training Step 8: Build generation model

3.12 Training Step 9: Create Configuration File

As a final step, a configuration file for the decoder is generated with all the correct paths for the generated model and a number of default parameter settings.

This file is called `model/moses.ini`

You will also need to train a language model. This is described in the decoder manual.

3.13 Building a Language Model

3.13.1 Content

- Language Models in Moses (page 89)
- Building a LM with the SRI LM Toolkit (page 90)
- On the IRST LM Toolkit (page 90)
- Building Huge Language Models (page 91)
- Binary Language Models (page 91)
- Quantized Language Models (page 92)
- Memory Mapping (page 92)
- Class Language Models and more (page 93)
- Chunk Language Models (page 94)
- RandLM (page 94)
- Installing RandLM (page 95)
- Building a randomized language model (page 95)
- Example 1: Building directly from corpora (page 96)
- Example 2: Building from an ARPA file (SRILM) (page 96)
- Example 3: Building a second randomized language model from the same data (page 97)
- Querying a randomized lanuage model (page 97)
- Building Randomised LMs using Billions of Words (page 98)

3.13.2 Language Models in Moses

The language model should be trained on a corpus that is suitable to the domain. If the translation model is trained on a parallel corpus, then the language model should be trained on the output side of that corpus, although using additional training data is often beneficial.

Our decoder works with the following language models:

- the SRI language modeling toolkit[1], which is freely available.
- the IRST language modeling toolkit[2], which is freely available and open source.
- the RandLM language modeling toolkit[3], which is freely available and open source.

In order to let Moses rely on one toolkit or on the other, it has to be compiled with the proper option:

[1] http://www.speech.sri.com/projects/srilm/
[2] http://sourceforge.net/projects/irstlm/
[3] http://sourceforge.net/projects/randlm/

- –with-srilm=<root dir of the SRILM toolkit>
- –with-irstlm=<root dir of the IRSTLM toolkit>
- –with-randlm=<root dir of the RandLM toolkit>

In the Moses configuration file, the type (SRI/IRST/RandLM) of the LM is specified through the first field (0/1/5 respectively) of the lines devoted to the specification of LMs:

```
0 <factor> <size> filename.srilm
```

or

```
1 <factor> <size> filename.irstlm
```

or

```
5 <factor> <size> filename.randlm
```

Both toolkits come with programs that create a language model file, as required by our decoder.

3.13.3 Building a LM with the SRI LM Toolkit

A language model can be created by calling:

```
% ngram-count -text CORPUS_FILE -lm SRILM_FILE
```

The command works also on compressed (`gz`) input and output. There are a variety of switches that can be used, we recommend `-interpolate -kndiscount`.

3.13.4 On the IRST LM Toolkit

Moses can also use language models created with the IRSTLM toolkit (see Federico & Cettolo, (ACL WS-SMT, 2007)[4]). The commands described in the following are supplied with the IRSTLM toolkit that has to be downloaded[5] and compiled separately.

IRSTLM toolkit handles LM formats which permit to reduce both storage and decoding memory requirements, and to save time in LM loading. In particular, it provides tools for:

- building (huge) LMs (page 91)
- quantizing LMs (page 92)

[4] http://www.aclweb.org/anthology-new/W/W07/W07-0712.pdf
[5] http://sourceforge.net/projects/irstlm

- compiling LMs (possibly quantized) into a binary format (page 91)
- accessing binary LMs through the memory mapping mechanism (page 92)
- query class and chunk LMs (page 93)

Building Huge Language Models

Training a language model from huge amounts of data can be definitively memory and time expensive. The IRSTLM toolkit features algorithms and data structures suitable to estimate, store, and access very large LMs. IRSTLM is open source and can be downloaded from here[6].

Typically, LM estimation starts with the collection of n-grams and their frequency counters. Then, smoothing parameters are estimated for each n-gram level; infrequent n-grams are possibly pruned and, finally, a LM file is created containing n-grams with probabilities and back-off weights. This procedure can be very demanding in terms of memory and time if applied to huge corpora. IRSTLM provides a simple way to split LM training into smaller and independent steps, which can be distributed among independent processes.

The procedure relies on a training script that makes little use of computer memory and implements the Witten-Bell smoothing method. (An approximation of the modified Kneser-Ney smoothing method is also available.) First, create a special directory `stat` under your working directory, where the script will save lots of temporary files; then, simply run the script *build-lm.sh* as in the example:

```
build-lm.sh -i "gunzip -c corpus.gz" -n 3 -o train.irstlm.gz -k 10
```

The script builds a 3-gram LM (option `-n`) from the specified input command (`-i`), by splitting the training procedure into 10 steps (`-k`). The LM will be saved in the output (`-o`) file train.irstlm.gz with an `intermediate ARPA` format. This format can be properly managed through the `compile-lm` command in order to produce a `compiled version` or a standard `ARPA version` of the LM.

For a detailed description of the procedure and of other commands available under IRSTLM please refer to the user manual supplied with the package.

Binary Language Models

You can convert your language model file (created either with the SRILM *ngram-count* command or with the IRSTLM toolkit) into a compact binary format with the command:

```
compile-lm language-model.srilm language-model.blm
```

Moses compiled with the IRSTLM toolkit is able to properly handle that binary format; the setting of moses.ini for that file is:

[6] http://sourceforge.net/projects/irstlm

```
1 0 3 language-model.blm
```

The binary format allows LMs to be efficiently stored and loaded. The implementation privileges memory saving rather than access time.

Quantized Language Models

Before compiling the language model, you can quantize (see Federico & Bertoldi, (ACL WS-SMT, 2006)[7]) its probabilities and back-off weights with the command:

```
quantize-lm language-model.srilm language-model.qsrilm
```

Hence, the binary format for this file is generated by the commmand:

```
compile-lm language-model.qsrilm language-model.qblm
```

The resulting language model requires less memory because all its probabilities and back-off weights are now stored in 1 byte instead of 4. No special setting of the configuration file is required: Moses compiled with the IRSTLM toolkit is able to read the necessary information from the header of the file.

Memory Mapping

It is possible to avoid the loading of the LM into the central memory by exploiting the memory mapping mechanism. Memory mapping permits the decoding process to directly access the (binary) LM file stored on the hard disk.

Warning: In case of parallel decoding in a cluster of computers, each process will access the same file. The possible large number of reading requests could overload the driver of the hard disk which the LM is stored on, and/or the network. One possible solution to such a problem is to store a copy of the LM on the local disk of each processing node, for example under the */tmp/* directory.

In order to activate the access through the memory mapping, simply add the suffix *.mm* to the name of the LM file (which must be stored in the binary format) and update the Moses configuration file accordingly.

As an example, let us suppose that the 3gram LM has been built and stored in binary format in the file

```
language-model.blm
```

Rename it for adding the *.mm* suffix:

```
mv language-model.blm  language-model.blm.mm
```

[7] http://www.aclweb.org/anthology/W/W06/W06-3113

or create a properly named symbolic link to the original file:

```
ln -s language-model.blm  language-model.blm.mm
```

Now, the activation of the memory mapping mechanism is obtained simply by updating the Moses configuration file as follows:

```
1 0 3 language-model.blm.mm
```

Class Language Models and more

Typically, LMs employed by Moses provide the probability of ngrams of single factors. In addition to the standard way, the IRSTLM toolkit allows Moses to query the LMs in other different ways. In the following description, it is assumed that the target side of training texts contains words which are concatenation of `N>=1` fields separated by the character `#`. Similarly to factored models, where the word is not anymore a simple token but a vector of factors that can represent different levels of annotation, here the word can be the concatenation of different tags for the surface form of a word, e.g.:

```
word#lemma#part-of-speech#word-class
```

Specific LMs for each tag can be queried by Moses simply by adding a fourth parameter in the line of the configuration file devoted to the specification of the LM. The additional parameter is a file containing (at least) the following header:

```
FIELD <int>
```

Possibly, it can also include a one-to-one map which is applied to each component of ngrams before the LM query:

```
w1 class(w1)
w2 class(w2)
...
wM class(wM)
```

The value of <int> determines the processing applied to the ngram components, which are supposed to be strings like `field0#field1#...#fieldN`:

- -1: the strings are used are they are; if the map is given, it is applied to the whole string before the LM query
- 0-9: the field number <int> is selected; if the map is given, it is applied to the selected field

- 00-99: the two fields corresponding to the two digits are selected and concatenated together using the character _ as separator. For example, if <int>=21, the LM is queried with ngrams of strings `field2_field1`. If the map is given, it is applied to the field corresponding to the first digit.

The last case is useful for lexicalization of LMs: if the fields n. 2 and 1 correspond to the POS and lemma of the actual word respectively, the LM is queried with ngrams of `POS_lemma`.

Chunk Language Models

A particular processing is performed whenever fields are supposed to correspond to *microtags*, i.e. the per-word projections of chunk labels. The processing aims at collapsing the sequence of microtags defining a chunk to the label of that chunk. The chunk LM is then queried with ngrams of chunk labels, in an asynchronous manner with respect to the sequence of words, as in general chunks consist of more words.

The collapsing operation is automatically activated if the sequence of microtags is:

```
(TAG TAG+ TAG+ ... TAG+ TAG)
```

or

```
TAG( TAG+ TAG+ ... TAG+ TAG)
```

Both those sequences are collapsed into a single chunk label (let us say `CHNK`) as long as `(TAG` / `TAG(`, `TAG+` and `TAG)` are all mapped into the same label `CHNK`. The map into different labels or a different use/position of characters `(`, `+` and `)` in the lexicon of tags prevent the collapsing operation.

Currently (Aug 2008), lexicalized chunk LMs are still under investigation and only non-lexicalized chunk LMs are properly handled; then, the range of admitted <`int`> values for this kind of LMs is -1...9, with the above described meaning.

3.13.5 RandLM

If you really want to build the largest LMs possible (for example, a 5-gram trained on several billions of words, such as the whole of the Gigaword Corpus) then you should look at the RandLM. This takes a very different approach to either the SRILM or the IRSTLM. It represents LMs using a randomized data structure (technically, variants of Bloom Filters). This can result in LMs that are ten times smaller than those created using the SRILM (and also smaller than IRSTLM), but at the cost of making decoding about four times slower.

If you use RandLM then it is recommended that you split your corpus of sentences into multiple blocks and run the decoder on each one in parallel. Since you are saving such a lot of memory, this will result in the same decoding speed as when using the SRILM and still benefitting from a reduced memory footprint.

Technical details of randomized language modelling can be found in a ACL paper (see Talbot and Osborne, (ACL 2007)[8])

Installing RandLM

RandLM is available at Sourceforge[9].

After extracting the tar ball, go to the directory `src` and type `make`.

For integrating RandLM into Moses, please see above.

Building a randomized language model

The `buildlm` binary (in `randlm/bin`) preprocesses and builds randomized language models.

The toolkit provides three ways for building a randomized language models:

1. from a tokenised corpus (this is useful for files around 100 million words or less)
2. from a precomputed backoff language model in ARPA format (this is useful if you want to use a precomputed SRILM model)
3. from a set of precomputed ngram-count pairs (this is useful if you need to build LMs from billions of words).

The former type of model will be referred to as a **CountRandLM** while the second will be referred to as a **BackoffRandLM**. Models built from precomputed ngram-count pairs are also of type "CountRandLM". CountRandLMs use either StupidBackoff or else Witten-Bell smoothing. BackoffRandLM models can use any smoothing scheme that the SRILM implements. Generally, CountRandLMs are smaller than BackoffRandLMs, but use less sophisticated smoothing. When using billions of words of training material there is less of a need for good smoothing and so CountRandLMs become appropriate.

The following parameters are important in all cases:

- `struct`: The randomized data structure used to represent the language model (currently only `BloomMap` and `LogFreqBloomFilter`).
- `order`: The order of the ngram model e.g., 3 for a trigram model.
- `falsepos`: The false positive rate of the randomized data structure on an inverse log scale so `-falsepos 8` produces a false positive rate of $1/2^8$.
- `values`: The quantization range used by the model. For a CountRandLM quantisation is performed by taking a logarithm. The base of the logarithm is set as $2^{1/\texttt{values}}$. For a BackoffRandLM a binning quantisation algorithm is used. The size of the codebook is set as $2^{\texttt{values}}$. A reasonable setting in both cases is `-values 8`.

[8] http://aclweb.org/anthology-new/P/P07/P07-1065.pdf
[9] http://sourceforge.net/projects/randlm/

- `input-path`: The location of data to be used to create the language model.
- `input-type`: The format of the input data. The following four formats are supported
  - for a CountRandLM:
    - `corpus` tokenised corpora one sentence per line;
    - `counts` ngram counts file (one count and one ngram per line);
  - Given a 'corpus' file the toolkit will create a 'counts' file which may be reused (see examples below).
  - for a BackoffRandLM:
    - `arpa` an ARPA backoff language model;
    - `backoff` language model file (two floats and one ngram per line).
  - Given an `arpa` file the toolkit will create a 'backoff' file which may be reused (see examples below).
- `output-prefix`:Prefix added to all output files during the construction of a randomized language model.

Example 1: Building directly from corpora

The command

```
./buildlm -struct BloomMap -falsepos 8 -values 8 -output-prefix model -order 3 < corpus
```

would produce the following files:-

```
model.BloomMap <- the randomized language model
model.counts.sorted <- ngram counts file
model.stats <- statistics file (counts of counts)
model.vcb   <- vocabulary file (not needed)
```

`model.BloomMap`: This randomized language model is ready to use on its own (see 'Querying a randomized language model' below).

`model.counts.sorted`: This is a file in the RandLM 'counts' format with one count followed by one ngram per line. It can be specified as shown in Example 3 below to avoid recomputation when building multiple randomized language models from the same corpus.

`model.stats`: This statistics file contains counts of counts and can be specified via the optional parameter '-statspath' as shown in Example 3 to avoid recomputation when building multiple randomized language models from the same data.

Example 2: Building from an ARPA file (SRILM)

The command

```
./buildlm -struct BloomMap -falsepos 8 -values 8 -output-prefix model -order 3
  -input-path precomputed.bo -input-type arpa
```

(where `precomputed.bo` contains an ARPA-formatted backoff model) would produce the following files:

```
model.BloomMap <- the randomized language model
model.backoff <- RandLM backoff file
model.stats <- statistics file (counts of counts)
model.vcb   <- vocabulary file (not needed)
```

`model.backoff` is a RandLM formatted copy of the ARPA model. It can be reused in the same manner as the 'model.counts.sorted' file (see Example 3).

Example 3: Building a second randomized language model from the same data

The command

```
./buildlm -struct BloomMap -falsepos 4 -values 8 -output-prefix model4 -order 3
  -input-path model.counts.sorted -input-type counts -stats-path model.stats
```

would construct a new randomized language model (`model4.BloomMap`) from the same data as used in Example 1 but with a different error rate (here `-falsepos 4`). This usage avoids re-tokenizing the corpus and recomputing the statistics file.

Querying a randomized lanuage model

Moses uses its own interface to the randLM, but it may be interesting to query the language model directly. The `querylm` binary (in `randlm/bin`) allows a randomized language model to be queried. Unless specified the scores provided by the tool will be conditional log probabilities.

The following parameters are available:-

- `randlm`: The path of the randomized language model built using the `buildlm` tool as described above.
- `test-path`: The location of test data to be scored by the model.
- `test-type`: The format of the test data: currently `corpus` and `ngrams` are supported. `corpus` will treat each line in the test file as a sentence and provide scores for all ngrams (adding <s> and </s>). `ngrams` will score each line once treating each as an independent ngram.
- `get-counts`: Return the counts of ngrams rather than conditional log probabilities (only supported by CountRandLM).
- `checks`: Applies sequential checks to ngrams to avoid unnecessary false positives.

Example: The command

```
./querylm -randlm model.BloomMap -test-path testfile -test-type ngrams -order 3 > scores
```

would write out conditional log probabilies one for each line in the file `test-file`.

Building Randomised LMs using Billions of Words

At some point you will discover that SRILM cannot build a LM using your data. RandLM natively uses a disk-based method for creating ngrams and counts, but this will be slow for large corpora. Instead you can create these ngram-count pairs using some other method. We use Hadoop to do this and tell RandLM to use this data instead, using these extra switches:

```
-keep-tmp-files -sorted-by-ngram -working-mem 5000 -input-type counts
```

This increases the amount of memory for sorting to 5GB. `counts` tells RandLM to expect data in the form of ngram-count pairs (see the previous example on the format required). Note that ngram-count pairs need to be sorted.

Another tip is to change the tmp dir:

```
-tmp-dir /path/to/tmp
```

Finally, there is a constant on the maximum count which needs increasing if you want to encode the Google 1T release:

```
-maxcount 50
```

3.14 Tuning

The training script `train-factored-model.perl` (page 75) produces a configuration file `moses.ini` which has default weights of questionable quality. That's why we need to obtain better weights by optimizing translation performance on a development set.

This is done with the tuning script `mert-moses-new.pl`. This new version of the minimum error rate trining script is based on a new C++ software. Details about the new implementations are given in Bertoldi, Haddow, Fouet, "Improved Minimum Error Rate Training in Moses", In Proc. of 3rd MT Marathon, Prague, Czech Republic. The new mert implementation is a standalone open-source software. The only interaction between Moses and the new software is given by the script `mert-moses-new.pl` itself.

This new implementation of mert stores feature scores and error statistics in separate files)possibly in a binary format) for each nbest-list (at each iteration), and use (some of) these files to optimize weights. At the moment weight optimization can be based on either BLEU or PER.

Most features of the old code `mert-moses.pl` are maintained, and some new ones are added.

The script are run as follows:

```
mert-moses-new.pl input-text references decoder-executable decoder.ini
```

Parameters:

- `input-text` and `references` are the development set, on which translation performance is optimized. The tuning script tries to find translations for `input-text` that resemble best the reference translations in `references`. The script works also with multiple output reference files, these have to be called `[references]0`, `[references]1`, `[references]2`, etc.
- `decoder-executable` is the location of the decoder binary to be used
- `decoder.ini` is the location of the configuration file to be used

Options:

- `-working-dir=STRING` (default `mert-dir`) directory that contains all files generated during the tuning process. Upon conclusion, it will contain a new `moses.ini` with better weights
- `-nbest=NUM` (default 100) size of n-best list to be generated at each run of the decoder
- `-jobs=NUM` if the script is run a cluster, this specifies how many jobs to submit (default: serial execution, does not use `qsub`)
- `-queue-flags=STRING` additional switches to pass to the parallelizer, eg. `'-qsub-prefix logname'`
- `-decoder-flags=STRING` additional parameters for the decoder
- `-lambdas=STRING` default values and ranges for lambdas, a complex string such as `'d:1,0.5-1.5 lm:1,0.5-1.5 ...'` (see below)
- `-average` use the average (not the default, closest) reference length as effective reference length for BLEU score computation
- `-closest` use the closest (default) reference length as effective reference length for BLEU score computation
- `-shortest` use the shortest (not the default, closest) reference length as effective reference length for BLEU score computation

- `-nocase` perform a case-insensitive evaluation between hypos and refs (default is false)
- `-activate-features=STRING` perform optimization on a specified subset of features (default is the optimization of all features); see below for details and for the correct syntax
- `-continue` continue the iterative optimization process from the last finished step (default is false); this is useful to recover a not-terminated optimization process (for example due to any system failure) without losing the first well-completed steps (see the note below for more details
- `-prev-aggregate-nbestlist=INT` number of previous steps to consider when loading data (default =-1). -1 means all previous, i.e. from iteration 1; 0 means no previous data, i.e. from actual iteration; 1 means 1 previous data , i.e. from the actual iteration and from the previous one; and so on.
- `-mertdir=STRING` path to the new implementation of mert software
- `-mertargs=STRING` extra arguments for mert, eg to specify score type (which is BLEU by default)
- `-help` gives a full list of options

Note: the optimized final weights are L1-normalized to 1 (i.e. sum_i |w_i| =1).

Note: the policy for computing the effective reference length in the BLEU score has changed (from revision 2461).

Note: the policy for case-sensitive/insensitive evaluation has changed (from revision 2461); now the default is case-sensitive.

Note: the `-continue` option relies on several files produced in the well-completed previous steps of the optimization process:

- "finished_step.txt"
- "runX.features.dat" for X=1,..,T
- "runX.scores.dat" for X=1,..,T
- "runT.weights.txt"
- "runT.mert.log"
- "runT.names.txt"

where T is the last well-completed step The file "finished_step.txt" should contain the value T.

Example: to store feature scores and error statistics in binary files and to use PER (instead of the default). Quotation marks are required.

```
--mertargs "--binary --sctype PER"
```

Example: to use only the nbest lists produced in the last 3 iterations of the mert process (plus the actual one):

```
--prev-aggregate-nbestlist=3
```

3.14.1 More on the lambda settings:

If you wish to optimize weights of all models your `moses.ini` mentions, and you want to use default values and intervals, you do not need to specify `-lambdas` at all.

`-lambdas=STRING` specifies the starting values and randomization ranges for the weights in a somwhat obstuse format. Each weight is specified as `start,min-max`, for instance `0.5,0.25-0.75` for a starting weight of 0.5 (used in the initial decoder run), and the randomized values between 0.25 and 0.75 during the parameter search. Weights have to be defined for reordering (`d`), language model (`lm`), translation model (`tm`), generation model (`g`), and word penalty (`w`), for instance by `d:1,0.5-1.5` for the reordering model. If there are multiple weights per component, these weights are specified in sequence separated by semicolons `;`.

Example: `d:1,0.5-1.5 lm:1,0.5-1.5 tm:0.3,0.25-0.75;0.2,0.25-0.75;0.2,0.25-0.75;0.3,0.25-0.75 w:0,-0.5-0.5` sets

- one weight for the distortion model, starting with 1, then randomized from 0.5-1.5
- one weight for the language model, starting with 1, then randomized from 0.5-1.5
- five weights for the translation model:
  - the first starting at 0.3, then randomized from 0.25-0.75
  - the first starting at 0.2, then randomized from 0.25-0.75
  - the first starting at 0.2, then randomized from 0.25-0.75
  - the first starting at 0.3, then randomized from 0.25-0.75
  - the first starting at 0, then randomized from -0.5 to 0.5
- one weight for the word penalty, starting with 0, then randomized from -0.5 to 0.5

3.14.2 Tuning on a subset of features

Sometimes it could be useful to optimize a subset of the feature weights. `mert-moses.pl` allows this through the parameter `-activate=list`, where `list` is a non-empty comma-separated list of features. Features are identified by `name_index`, where `name` is the group name and `index` is the position of the feature inside the group. The group name are:

```
d   distortion model
lm  language models
tm  translation models
w   word penalty
I   posterior probability for confusion network
```

and the index starts from 0; the index is mandatory even if only one feature occurs in a group. If no features are specified (`-activate-features=”`) or the parameter `-activate-features` is not set, `mert-moses-new.pl` perform optimization over all available features.

For instance, setting the option as follows

```
--activate-features=d_0,d_4,lm_0,tm_3,w_0
```

only the following features will be optimized: the first (d_0) and the fifth (d_4) distortion model weights, the first language model weight (lm_0), the fourth (tm_3) translation model weight, and the first (w_0) word penalty.

IMPORTANT:

- The configuration file used for the first iteration takes the feature weights from the `-lambdas` parameter (or from their defaults), and not from the configuration file passed through the command line. Hence, if you want to assign specific (already optimized) values for NOT-activated features, please set them by means of the `-lambdas` parameter. For these NOT-activated features, you must also specify their ranges in the `-lambdas` parameter to maintain the correct syntax although they are not exploited.
- Please, pay attention that `mert-moses-new.pl` fails if a wrong feature is specified; for instance lm_2 is not allowed if only one or two language models are used.
- There are some differences with respect to the old version of the script (`mert-moses.pl`):
  - you can no more specify a group of features
  - features are kept fixed to their initial values (specified with `-lambdas`)

3.14.3 Tuning on a subset of features (Old version)

`mert-moses.pl` allows this through the parameter `-activate=list`, where `list` is a non-empty comma-separated list of features. Features are identified by their names. The feature names can be found in the nbest list, in the file names.txt in the working directory during the minimum error training, and in the moses help (`moses -help`). Main features are:

```
d   distortion model
lm  language models
tm  translation models
w   word penalty
I   posterior probability for confusion network
```

The parameter `-activate=list` activates the optimization of only the listed feature weights. The ratios among the remaining ones are fixed, and are taken from the `-lambdas` parameter.

If a feature have more scores (eg. `tm` with 5 scores), optimization can be performed on:

- all of them (`-activate=tm`)
- any of them by specifying its index (`-activate=tm_2,tm_3`)

The optimization of a subset of feature weights works as follows:

- not-active features are weighted summed to create an extra feature.
- the active features and this extra feature are optimized (and normalized)
- the initial not-active weights are multiplied by the optimal weight of the extra feature
- the full set of weights is normalized to 1.

Note that if parameter `-activate` is not set, all weights are optimized.

Example: Features are: `d lm tm tm tm tm tm w`

Parameters are set as follows: `-lambdas="d:1,0.5-1.5 lm:1,0.5-1.5 tm:0.3,0.25-0.75;0.2,0.25-0.75;0 w:0,-0.5-0.5" -activate=d,tm_1,tm_5`

Features `d`, the first `tm` and the fifth `tm` will be optimized. Features `lm`, the second, third and fourth `tm` and `w` will NOT be optimized. The ratios among them comes from their initial values: 1, 0.2, 0.2, 0.3, and 0, respectively.

If optimal weights are 0.11, 0.54, and 0.09 respectively, and 0.26 for the extra feature, the final set of weights is

```
0.11 0.26(=0.26*1)
0.54 0.052(=0.26*0.2)
0.052(=0.26*0.2)
0.078(=0.26*0.3)
0.09 0(=0.26*0)
```

and after normalization

```
0.093 0.220 0.457 0.044 0.044 0.066 0.076 0
```

You can also use the old version of MERT script `mert-moses.pl`, but some feature are not available.

3.14.4 Running the script on a cluster

If the script is run as a multi-job process (`-jobs`) on a Grid Engine[10] cluster, you will run the script on the head node. The script submits all compute-heavy parts as jobs to the cluster. In other words, you do not submit the script itself as a job to the cluster.

[10] http://gridengine.sunsource.net/

4

Background

4.1 Background

Statistical Machine Translation as a research area started in the late 1980s with the Candide project at IBM. IBM's original approach maps individual words to words and allows for deletion and insertion of words.

Lately, various researchers have shown better translation quality with the use of phrase translation. Phrase-based MT can be traced back to Och's alignment template model, which can be re-framed as a phrase translation system. Other researchers used augmented their systems with phrase translation, such as Yamada, who use phrase translation in a syntax-based model.

Marcu introduced a joint-probability model for phrase translation. At this point, most competitive statistical machine translation systems use phrase translation, such as the CMU, IBM, ISI, and Google systems, to name just a few. Phrase-based systems came out ahead at a recent international machine translation competition (DARPA TIDES Machine Translation Evaluation 2003-2006 on Chinese-English and Arabic-English).

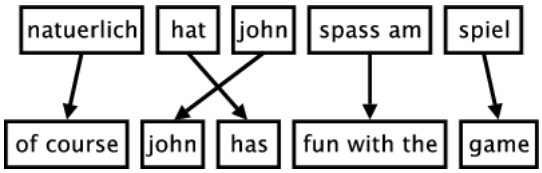
Of course, there are other ways to do machine translation. Most commercial systems use transfer rules and a rich translation lexicon. Until recently, machine translation research has been focused on knowledge based systems that use a interlingua representation as an intermediate step between input and output.

There are also other ways to do statistical machine translation. There is some effort in building syntax-based models that either use real syntax trees generated by syntactic parsers, or tree transfer methods motivated by syntactic reordering patterns.

The phrase-based statistical machine translation model we present here was defined by [Koehn/Och/Marcu, 2003]. See also the description by [Zens, 2002]. The alternative phrase-based methods differ in the way the phrase translation table is created, which we disuss in detail below.

4.1.1 Model

The figure below illustrates the process of phrase-based translation. The input is segmented into a number of sequences of consecutive words (so-called *phrases*). Each phrase is translated into an English phrase, and English phrases in the output may be reordered.



In this section, we will define the phrase-based machine translation model formally. The phrase translation model is based on the noisy channel model. We use Bayes rule to reformulate the translation probability for translating a foreign sentence **f** into English **e** as

$\text{argmax}_{\mathbf{e}} p(\mathbf{e}|\mathbf{f}) = \text{argmax}_{\mathbf{e}}(\mathbf{f}|\mathbf{e})\ (\mathbf{e})$

This allows for a language model **e** and a separate translation model p(**f**|**e**).

During decoding, the foreign input sentence **f** is segmented into a sequence of I phrases $\bar{f}_1^{\,\mathrm{I}}$. We assume a uniform probability distribution over all possible segmentations.

Each foreign phrase $\bar{f}_i$ in $\bar{f}_1^{\,\mathrm{I}}$ is translated into an English phrase $\bar{e}_i$. The English phrases may be reordered. Phrase translation is modeled by a probability distribution $\phi(\bar{f}_i|\bar{e}_i)$. Recall that due to the Bayes rule, the translation direction is inverted from a modeling standpoint.

Reordering of the English output phrases is modeled by a relative distortion probability distribution $d(\text{start}_i, \text{end}_{i-1})$, where start_i denotes the start position of the foreign phrase that was translated into the ith English phrase, and end_{i-1} denotes the end position of the foreign phrase that was translated into the *(i-1)*th English phrase.

We use a simple distortion model $d(\text{start}_i, \text{end}_{i-1}) = \alpha^{|\text{start}_i - \text{end}_{i-1} - 1|}$ with an appropriate value for the parameter α.

In order to calibrate the output length, we introduce a factor ω (called word cost) for each generated English word in addition to the trigram language model p_{LM}. This is a simple means to optimize performance. Usually, this factor is larger than 1, biasing toward longer output.

In summary, the best English output sentence $\mathbf{e}_{\text{best}}$ given a foreign input sentence **f** according to our model is

$\mathbf{e}_{\text{best}} = \text{argmax}_{\mathbf{e}}(\mathbf{e}|\mathbf{f}) = \text{argmax}_{\mathbf{e}}(\mathbf{f}|\mathbf{e})\ p_{\text{LM}}(\mathbf{e})\ \omega^{\text{length}(\mathbf{e})}$

where p(**f**|**e**) is decomposed into

$p(\bar{f}_1^{\,\mathrm{I}}|\bar{e}_1^{\,\mathrm{I}}) = \Phi_{i=1}^{\mathrm{I}}\ \phi(\bar{f}_i|\bar{e}_i)\ d(\text{start}_i, \text{end}_{i-1})$

4.1.2 Word Alignment

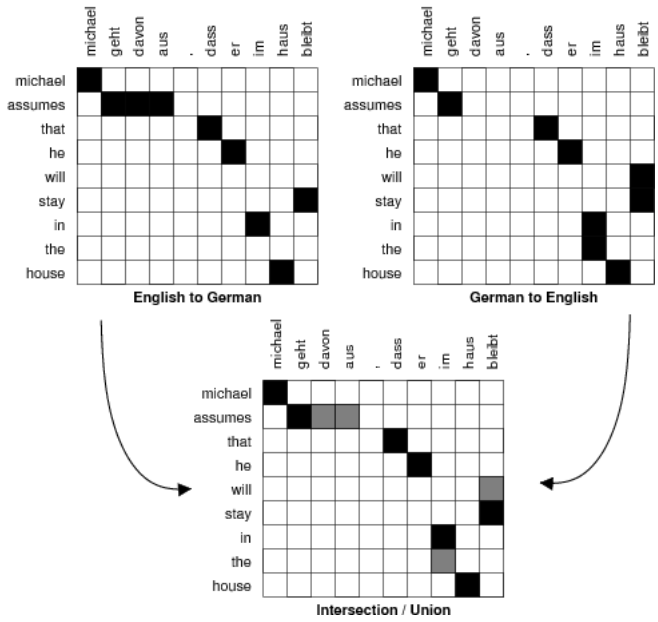
When describing the phrase-based translation model so far, we did not discuss how to obtain the model parameters, especially the phrase probability translation table that maps foreign

phrases to English phrases.

Most recently published methods on extracting a phrase translation table from a parallel corpus start with a word alignment. Word alignment is an active research topic. For instance, this problem was the focus as a shared task at a recent data driven machine translation workshop[1]. See also the systematic comparison by Och and Ney (Computational Linguistics, 2003).

At this point, the most common tool to establish a word alignment is to use the toolkit GIZA++[2]. This toolkit is an implementation of the original IBM Models that started statistical machine translation research. However, these models have some serious draw-backs. Most importantly, they only allow at most one English word to be aligned with each foreign word. To resolve this, some transformations are applied.

First, the parallel corpus is aligned bidirectionally, e.g., Spanish to English and English to Spanish. This generates two word alignments that have to be reconciled. If we intersect the two alignments, we get a high-precision alignment of high-confidence alignment points. If we take the union of the two alignments, we get a high-recall alignment with additional alignment points. See the figure below for an illustration.



Researchers differ in their methods where to go from here. We describe the details below.

4.1.3 Methods for Learning Phrase Translations

Most of the recently proposed methods use a word alignment to learn a phrase translation table. We discuss three such methods in this section and one exception.

Marcu and Wong

[1] http://www.statmt.org/wpt05/

[2] http://www.isi.edu/ och/GIZA++.html

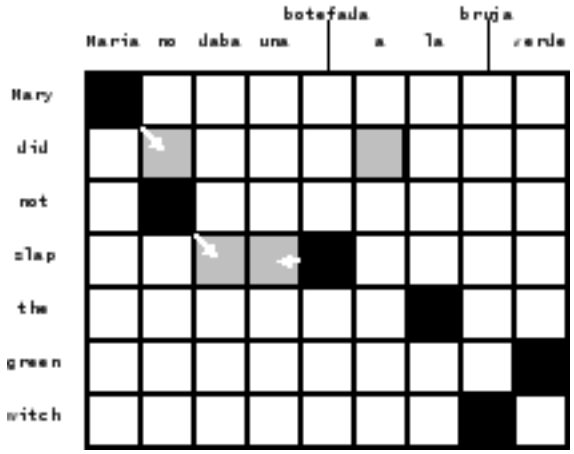
First, the exception: Marcu and Wong (EMNLP, 2002) proposed to establish phrase correspondences directly in a parallel corpus. To learn such correspondences, they introduced a phrase-based joint probability model that simultaneously generates both the source and target sentences in a parallel corpus.

Expectation Maximization learning in Marcu and Wong's framework yields both (i) a joint probability distribution $\phi(\bar{e}, \bar{f})$, which reflects the probability that phrases $\bar{e}$ and $\bar{f}$ are translation equivalents; (ii) and a joint distribution *d(i,j)*, which reflects the probability that a phrase at position *i* is translated into a phrase at position *j*.

To use this model in the context of our framework, we simply marginalize the joint probabilities estimated by Marcu and Wong (EMNLP, 2002) to conditional probabilities. Note that this approach is consistent with the approach taken by Marcu and Wong themselves, who use conditional models during decoding.

4.1.4 Och and Ney

Och and Ney (Computational Linguistics, 2003) propose a heuristic approach to refine the alignments obtained from Giza++. At a minimum, all alignment points of the intersection of the two alignments are maintained. At a maximum, the points of the union of the two alignments are considered. To illustrate this, see the figure below. The intersection points are black, the additional points in the union are shaded grey.



Och and Ney explore the space between intersection and union with expansion heuristics that start with the intersection and add additional alignment points. The decision which points to add may depend on a number of criteria:

- In which alignment does the potential alignment point exist? Foreign-English or English-foreign?
- Does the potential point neighbor already established points?

- Does *neighboring* mean directly adjacent (block-distance), or also diagonally adjacent?
- Is the English or the foreign word that the potential point connects unaligned so far? Are both unaligned?
- What is the lexical probability for the potential point?

Och and Ney (Computational Linguistics, 2003) are ambigous in their description about which alignment points are added in their refined method. We reimplemented their method for Moses, so we will describe this interpretation here.

Our heuristic proceeds as follows: We start with intersection of the two word alignments. We only add new alignment points that exist in the union of two word alignments. We also always require that a new alignment point connects at least one previously unaligned word.

First, we expand to only directly adjacent alignment points. We check for potential points starting from the top right corner of the alignment matrix, checking for alignment points for the first English word, then continue with alignment points for the second English word, and so on.

This is done iteratively until no alignment point can be added anymore. In a final step, we add non-adjacent alignment points, with otherwise the same requirements.

We collect all aligned phrase pairs that are consistent with the word alignment: The words in a legal phrase pair are only aligned to each other, and not to words outside. The set of bilingual phrases **BP** can be defined formally (Zens, KI 2002) as:

$$BP(f_1^J, e_1^J, A) = \{(f_j^{j+m}, e_i^{i+n})\} : \forall (i', j') \in A : j <= j' <= j+m \leftrightarrow i <= i' <= i+m$$

The figure below displays all the phrase pairs that are collected according to this definition for the alignment from our running example.



(Maria, Mary), (no, did not), (slap, daba una bofetada), (a la, the), (bruja, witch), (verde, green), (Maria no, Mary did not), (no daba una bofetada, did not slap), (daba una bofetada a la, slap the), (bruja verde, green witch), (Maria no daba una bofetada, Mary did not slap), (no daba una bofetada a la, did not slap the), (a la bruja verde, the green witch) (Maria no daba una bofetada a la, Mary did not slap the), (daba una bofetada a la bruja verde, slap the green witch), (no daba una bofetada a la bruja verde, did not slap the green witch), (Maria no daba una bofetada a la bruja verde, Mary did not slap the green witch)

Given the collected phrase pairs, we estimate the phrase translation probability distribution by relative frequency: $\phi(\bar{f}|\bar{e})$ = count($\bar{f}$,$\bar{e}$) $\Sigma_{\bar{f}}$ count($\bar{f}$,$\bar{e}$)

No smoothing is performed, although lexical weighting addresses the problem of sparse data. For more details, see our paper on phrase-based translation (Koehn at al, HLT-NAACL 2003).

Tillmann

Tillmann (EMNLP, 2003) proposes a variation of this method. He starts with phrase alignments based on the intersection of the two Giza alignments and uses points of the union to expand these. See his presentation for details.

Venugopal, Zhang, and Vogel

Venugopal et al. (ACL 2003) allows also for the collection of phrase pairs that are violated by the word alignment. They introduce a number of scoring methods take consistency with the word alignment, lexical translation probabilities, phrase length, etc. into account.

Zhang et al. (2003) proposes a phrase alignment method that is based on word alignments and tries to find a unique segmentation of the sentence pair, as it is done by Marcu and Wong directly. This enables them to estimate joint probability distributions, which can be marginalized into conditional probability distributions.

Vogel et al. (2003) reviews these two methods and shows that the combining phrase tables generated by different methods improves results.

4.2 Decoder

This section describes the Moses decoder from a more theoretical perspective. The decoder was originally developed for the phrase model proposed by Marcu and Wong. At that time, only a greedy hill-climbing decoder was available, which was unsufficent for our work on noun phrase translation (Koehn, PhD, 2003).

The decoder implements a beam search and is roughly similar to work by Tillmann (PhD, 2001) and Och (PhD, 2002). In fact, by reframing Och's alignment template model as a phrase translation model, the decoder is also suitable for his model, as well as other recently proposed phrase models.

We start this section with defining the concept of translation options, describe the basic mechanism of beam search, and its neccessary components: pruning, future cost estimates. We conclude with background on n-best list generation.

4.2.1 Translation Options

Given an input string of words, a number of phrase translations could be applied. We call each such applicable phrase translation a *translation option*. This is illustrated in the figure below, where a number of phrase translations for the Spanish input sentence *Maria no daba uma bofetada a la bruja verde* are given.



These translation options are collected before any decoding takes place. This allows a quicker lookup than consulting the whole phrase translation table during decoding. The translation options are stored with the information

- first foreign word covered
- last foreign word covered
- English phrase translation
- phrase translation probability

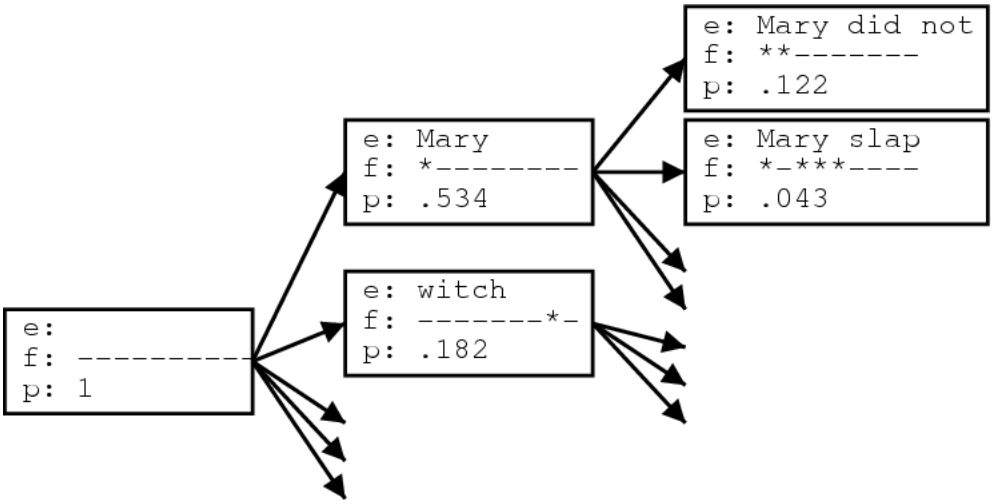
Note that only the translation options that can be applied to a given input text are necessary for decoding. Since the entire phrase translation table may be to big to fit into memory, we can restrict ourselves to these translation options to overcome such computational concerns. We may even generate a phrase translation table on demand that only includes valid translation options for a given input text. This way, a full phrase translation table (that may be computationally too expensive to produce) may never have to be built.

4.2.2 Core Algorithm

The phrase-based decoder we developed employs a beam search algorithm, similar to the one used by Jelinek (book "Statistical Methods for Speech Recognition", 1998) for speech recognition. The English output sentence is generated left to right in form of hypotheses.

This process illustrated in the figure below. Starting from the initial hypothesis, the first expansion is the foreign word *Maria*, which is translated as *Mary*. The foreign word is marked

as translated (marked by an asterisk). We may also expand the initial hypothesis by translating the foreign word *bruja* as *witch*.



We can generate new hypotheses from these expanded hypotheses. Given the first expanded hypothesis we generate a new hypothesis by translating *no* with *did not*. Now the first two foreign words *Maria* and *no* are marked as being covered. Following the back pointers of the hypotheses we can read of the (partial) translations of the sentence.

Let us now describe the beam search more formally. We begin the search in an initial state where no foreign input words are translated and no English output words have been generated. New states are created by extending the English output with a phrasal translation of that covers some of the foreign input words not yet translated.

The current cost of the new state is the cost of the original state multiplied with the translation, distortion and language model costs of the added phrasal translation. Note that we use the informal concept *cost* analogous to probability: A high cost is a low probability.

Each search state (hypothesis) is represented by

- a back link to the best previous state (needed for find the best translation of the sentence by back-tracking through the search states)
- the foreign words covered so far
- the last two English words generated (needed for computing future language model costs)
- the end of the last foreign phrase covered (needed for computing future distortion costs)
- the last added English phrase (needed for reading the translation from a path of hypotheses)
- the cost so far

- an estimate of the future cost (is precomputed and stored for efficiency reasons, as detailed in below in special section)

Final states in the search are hypotheses that cover all foreign words. Among these the hypothesis with the lowest cost (highest probability) is selected as best translation.

The algorithm described so far can be used for exhaustively searching through all possible translations. In the next sections we will describe how to optimize the search by discarding hypotheses that cannot be part of the path to the best translation. We then introduce the concept of comparable states that allow us to define a beam of good hypotheses and prune out hypotheses that fall out of this beam. In a later section, we will describe how to generate an (approximate) n-best list.

4.2.3 Recombining Hypotheses

Recombining hypothesis is a risk-free way to reduce the search space. Two hypotheses can be recombined if they agree in

- the foreign words covered so far
- the last two English words generated
- the end of the last foreign phrase covered

If there are two paths that lead to two hypotheses that agree in these properties, we keep only the cheaper hypothesis, e.g., the one with the least cost so far. The other hypothesis cannot be part of the path to the best translation, and we can safely discard it.

Note that the inferior hypothesis can be part of the path to the second best translation. This is important for generating n-best lists.

4.2.4 Beam Search

While the recombination of hypotheses as described above reduces the size of the search space, this is not enough for all but the shortest sentences. Let us estimate how many hypotheses (or, states) are generated during an exhaustive search. Considering the possible values for the properties of unique hypotheses, we can estimate an upper bound for the number of states by $N \quad 2^{n_f} \ |V_e|^2 \ n_f$ where n_f is the number of foreign words, and $|V_e|$ the size of the English vocabulary. In practice, the number of possible English words for the last two words generated is much smaller than $|V_e|^2$. The main concern is the exponential explosion from the 2^{n_f} possible configurations of foreign words covered by a hypothesis. Note this causes the problem of machine translation to become NP-complete (Knight, Computational Linguistics, 1999) and thus dramatically harder than, for instance, speech recognition.

In our beam search we compare the hypotheses that cover the same *number* of foreign words and prune out the inferior hypotheses. We could base the judgment of what inferior hypotheses are on the cost of each hypothesis so far. However, this is generally a very bad criterion, since it biases the search to first translating the easy part of the sentence. For instance, if there is a three word foreign phrase that easily translates into a common English phrase, this may carry much less cost than translating three words separately into uncommon English words. The search will prefer to start the sentence with the easy part and discount alternatives too early.

So, our measure for pruning out hypotheses in our beam search does not only include the cost so far, but also an estimate of the future cost. This future cost estimation should favor hypotheses that already covered difficult parts of the sentence and have only easy parts left, and discount hypotheses that covered the easy parts first. We describe the details of our future cost estimation in the next section.

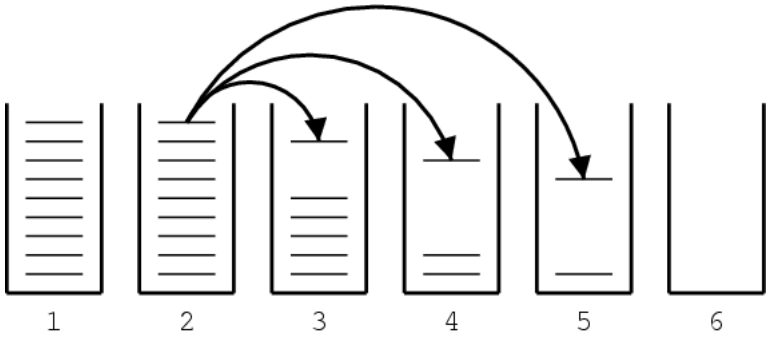
Given the cost so far and the future cost estimation, we can prune out hypotheses that fall outside the beam. The beam size can be defined by threshold and histogram pruning. A relative threshold cuts out a hypothesis with a probability less than a factor α of the best hypotheses (e.g., $\alpha = 0.001$). Histogram pruning keeps a certain number n of hypotheses (e.g., $n = 100$).

Note that this type of pruning is not risk-free (opposed to the recombination, which we described earlier). If the future cost estimates are inadequate, we may prune out hypotheses on the path to the best scoring translation. In a particular version of beam search, A* search, the future cost estimate is required to be _admissible_, which means that it never overestimates the future cost. Using best-first search and an admissible heuristic allows pruning that is risk-free. In practice, however, this type of pruning does not sufficiently reduce the search space. See more on search in any good Artificial Intelligence text book, such as the one by Russel and Norvig ("Artificial Intelligence: A Modern Approach").

The figure below gives pseudo-code for the algorithm we used for our beam search. For each number of foreign words covered, a hypothesis stack in created. The initial hypothesis is placed in the stack for hypotheses with no foreign words covered. Starting with this hypothesis, new hypotheses are generated by committing to phrasal translations that covered previously unused foreign words. Each derived hypothesis is placed in a stack based on the number of foreign words it covers.

```
initialize hypothesisStack[0 .. nf];
create initial hypothesis hyp_init;
add to stack hypothesisStack[0];
for i=0 to nf-1:
  for each hyp in hypothesisStack[i]:
    for each new_hyp that can be derived from hyp:
      nf[new_hyp] = number of foreign words covered by new_hyp;
      add new_hyp to hypothesisStack[nf[new_hyp]];
      prune hypothesisStack[nf[new_hyp]];
find best hypothesis best_hyp in hypothesisStack[nf];
output best path that leads to best_hyp;
```

We proceed through these hypothesis stacks, going through each hypothesis in the stack, deriving new hypotheses for this hypothesis and placing them into the appropriate stack (see figure below for an illustration). After a new hypothesis is placed into a stack, the stack may have to be pruned by threshold or histogram pruning, if it has become too large. In the end, the best hypothesis of the ones that cover all foreign words is the final state of the best translation. We can read off the English words of the translation by following the back links in each hypothesis.



4.2.5 Future Cost Estimation

Recall that for excluding hypotheses from the beam we do not only have to consider the cost so far, but also an estimate of the future cost. While it is possible to calculate the cheapest possible future cost for each hypothesis, this is computationally so expensive that it would defeat the purpose of the beam search.

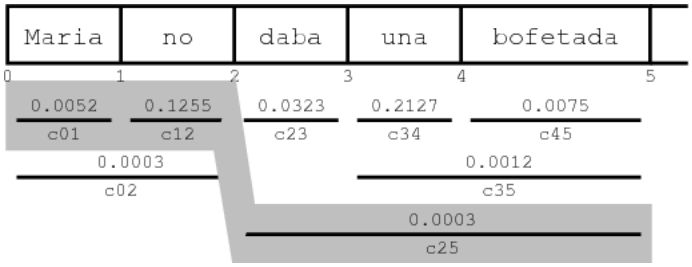
The future cost is tied to the foreign words that are not yet translated. In the framework of the phrase-based model, not only may single words be translated individually, but also consecutive sequences of words as a phrase.

Each such translation operation carries a translation cost, language model costs, and a distortion cost. For our future cost estimate we consider only translation and language model costs. The language model cost is usually calculated by a trigram language model. However, we do not know the preceding English words for a translation operation. Therefore, we approximate this cost by computing the language model score for the generated English words alone. That means, if only one English word is generated, we take its unigram probability. If two words are generated, we take the unigram probability of the first word and the bigram probability of the second word, and so on.

For a sequence of foreign words multiple overlapping translation options exist. We just described how we calculate the cost for each translation option. The cheapest way to translate the sequence of foreign words includes the cheapest translation options. We approximate the cost for a path through translation options by the product of the cost for each option.

To illustrate this concept, refer to the figure below. The translation options cover different consecutive foreign words and carry an estimated cost c_{ij}. The cost of the shaded path through

the sequence of translation options is $c_{01}c_{12}c_{25} = 1.9578 * 10^{-7}$.



The cheapest path for a sequence of foreign words can be quickly computed with dynamic programming. Also note that if the foreign words not covered so far are two (or more) disconnected sequences of foreign words, the combined cost is simply the product of the costs for each contiguous sequence. Since there are only *n(n+1)/2* contiguous sequences for *n* words, the future cost estimates for these sequences can be easily precomputed and cached for each input sentence. Looking up the future costs for a hypothesis can then be done very quickly by table lookup. This has considerable speed advantages over computing future cost on the fly.

4.2.6 N-Best Lists Generation

Usually, we expect the decoder to give us the best translation for a given input according to the model. But for some applications, we might also be interested in the second best translation, third best translation, and so on.

A common method in speech recognition, that has also emerged in machine translation is to first use a machine translation system such as our decoder as a base model to generate a set of candidate translations for each input sentence. Then, additional features are used to rescore these translations.

An n-best list is one way to represent multiple candidate translations. Such a set of possible translations can also be represented by word graphs (Ueffing et al., EMNLP 2002) or forest structures over parse trees (Langkilde, EACL 2002). These alternative data structures allow for more compact representation of a much larger set of candidates. However, it is much harder to detect and score global properties over such data structures.

Additional Arcs in the Search Graph

Recall the process of state expansions. The generated hypotheses and the expansions that link them form a graph. Paths branch out when there are multiple translation options for a hypothesis from which multiple new hypotheses can be derived. Paths join when hypotheses are recombined.

Usually, when we recombine hypotheses, we simply discard the worse hypothesis, since it cannot possibly be part of the best path through the search graph (in other words, part of the best translation).

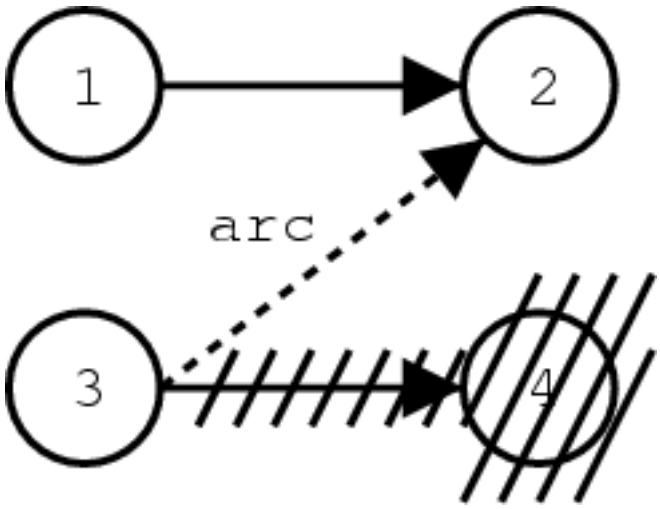
But since we are now also interested in the second best translation, we cannot simply discard information about that hypothesis. If we would do this, the search graph would only contain one path for each hypothesis in the last hypothesis stack (which contains hypotheses that cover all foreign words).

If we store information that there are multiple ways to reach a hypothesis, the number of possible paths also multiplies along the path when we traverse backward through the graph.

In order to keep the information about merging paths, we keep a record of such merges that contains

- identifier of the previous hypothesis
- identifier of the lower-cost hypothesis
- cost from the previous to higher-cost hypothesis

The figure below gives an example for the generation of such an arc: in this case, the hypotheses 2 and 4 are equivalent in respect to the heuristic search, as detailed above. Hence, hypothesis 4 is deleted. But since we want keep the information about the path leading from hypothesis 3 to 2, we store a record of this arc. The arc also contains the cost added from hypothesis 3 to 4. Note that the cost from hypothesis 1 to hypothesis 2 does not have to be stored, since it can be recomputed from the hypothesis data structures.



Mining the Search Graph for an n-Best List

The graph of the hypothesis space can be also be viewed as a probabilistic finite state automaton. The hypotheses are states, and the records of back-links and the additionally stored arcs are state transitions. The added probability scores when expanding a hypothesis are the costs of the state transitions.

Finding the n-best path in such a probabilistic finite state automaton is a well-studied problem. In our implementation, we store the information about hypotheses, hypothesis transitions,

and additional arcs in a file that can be processed by the finite state toolkit Carmel[3], which we use to mine the n-best lists. This toolkit uses the _n_ shortest paths algorithm by Eppstein (FOCS, 1994).

Our method is related to work by Ueffing (2002) for generating n-best lists for IBM Model 4.

4.3 Factored Translation Models

The current state-of-the-art approach to statistical machine translation, so-called phrase-based models, are limited to the mapping of small text chunks (phrases) without any explicit use of linguistic information, may it be morphological, syntactic, or semantic. Such additional information has been demonstrated to be valuable by integrating it in pre-processing or post-processing.

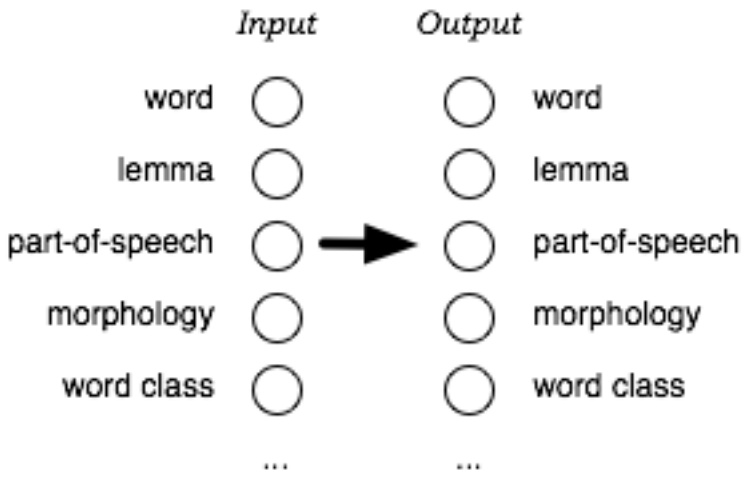
However, a tighter integration of linguistic information into the translation model is desirable for two reasons:

- Translation models that operate on more general representations, such as lemmas instead of surface forms of words, can draw on richer statistics and overcome the data sparseness problems caused by limited training data.

- Many aspects of translation can be best explained on a morphological, syntactic, or semantic level. Having such information available to the translation model allows the direct modeling of these aspects. For instance: reordering at the sentence level is mostly driven by general syntactic principles, local agreement constraints show up in morphology, etc.

Therefore, we developed a framework for statistical translation models that tightly integrates additional information. Our framework is an extension of the phrase-based approach. It adds additional annotation at the word level. A word in our framework is not anymore only a token, but a vector of factors that represent different levels of annotation (see figure below).

[3] http://www.isi.edu/licensed-sw/carmel/



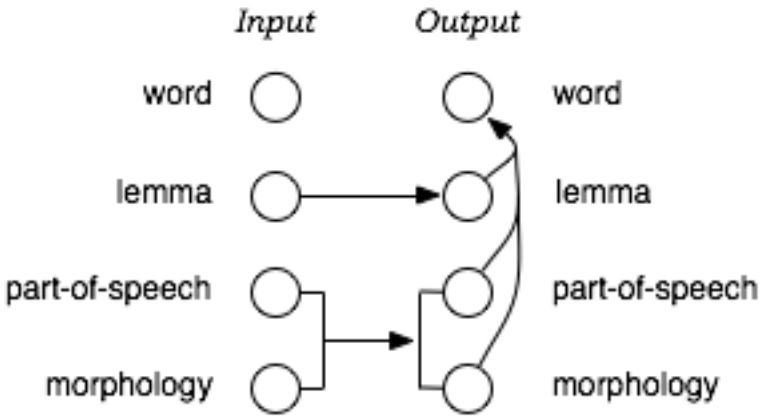
4.3.1 Motivating Example: Morphology

One example to illustrate the short-comings of the traditional surface word approach in statistical machine translation is the poor handling of morphology. Each word form is treated as a token in itself. This means that the translation model treats, say, the word *house* completely independent of the word *houses*. Any instance of *house* in the training data does not add any knowledge to the translation of *houses*.

In the extreme case, while the translation of *house* may be known to the model, the word *houses* may be unknown and the system will not be able to translate it. While this problem does not show up as strongly in English - due to the very limited morphological production in English - it does constitute a significant problem for morphologically rich languages such as Arabic, German, Czech, etc.

Thus, it may be preferably to model translation between morphologically rich languages on the level of lemmas, and thus pooling the evidence for different word forms that derive from a common lemma. In such a model, we would want to translate lemma and morphological information separately, and combine this information on the output side to ultimately generate the output surface words.

Such a model can be defined straight-forward as a factored translation model. See figure below for an illustration of this model in our framework.



Note that while we illustrate the use of factored translation models on such a linguistically motivated example, our framework also applies to models that incorporate statistically defined word classes, or any other annotation.

4.3.2 Decomposition of Factored Translation

The translation of factored representations of input words into the factored representations of output words is broken up into a sequence of **mapping steps** that either **translate** input factors into output factors, or **generate** additional output factors from existing output factors.

Recall the example of a factored model motivated by morphological analysis and generation. In this model the translation process is broken up into the following three mapping steps:

- **Translate** input lemmas into output lemmas
- **Translate** morphological and POS factors
- **Generate** surface forms given the lemma and linguistic factors

Factored translation models build on the phrase-based approach, which defines a segmentation of the input and output sentences into phrases. Our current implementation of factored translation models follows strictly the phrase-based approach, with the additional decomposition of phrase translation into a sequence of mapping steps. Since all mapping steps operate on the same phrase segmentation of the input and output sentence into phrase pairs, we call these **synchronous factored models**.

Let us now take a closer look at one example, the translation of the one-word phrase *h"auser* into English. The representation of *h"auser* in German is: surface-form *h"auser* | lemma *haus* | part-of-speech *NN* | count *plural* | case *nominative* | gender *neutral*.

The three mapping steps in our morphological analysis and generation model may provide the following applicable mappings:

- **Translation:** Mapping lemmas
  - *haus -> house, home, building, shell*
- **Translation:** Mapping morphology
  - *NN|plural-nominative-neutral -> NN|plural, NN|singular*
- **Generation:** Generating surface forms
  - *house|NN|plural -> houses*
  - *house|NN|singular -> house*
  - *home|NN|plural -> homes*
  - ...

We call the application of these mapping steps to an input phrase **expansion**. Given the multiple choices for each step (reflecting the ambiguity in translation), each input phrase may be expanded into a list of translation options. The German *h"auser|haus|NN|plural-nominative-neutral* may be expanded as follows:

- **Translation:** Mapping lemmas
  - { *?|house|?|?, ?|home|?|?, ?|building|?|?, ?|shell|?|?* }
- **Translation:** Mapping morphology
  - { *?|house|NN|plural, ?|home|NN|plural, ?|building|NN|plural, ?|shell|NN|plural, ?|house|NN|singular,...* }
- **Generation:** Generating surface forms
  - { *houses|house|NN|plural, homes|home|NN|plural, buildings|building|NN|plural, shells|shell|NN|plural, house|house|NN|singular, ...* }

4.3.3 Statistical Model

Factored translation models follow closely the statistical modeling approach of phrase-based models (in fact, phrase-based models are a special case of factored models). The main difference lies in the preparation of the training data and the type of models learned from the data.

Training

The training data (a parallel corpus) has to be annotated with the additional factors. For instance, if we want to add part-of-speech information on the input and output side, we need to obtain part-of-speech tagged training data. Typically this involves running automatic tools on the corpus, since manually annotated corpora are rare and expensive to produce.

Next, we need to establish a word-alignment for all the sentences in the parallel training corpus. Here, we use the same methodology as in phrase-based models (symmetrized GIZA++ alignments). The word alignment methods may operate on the surface forms of words, or on any of the other factors. In fact, some preliminary experiments have shown that word alignment based on lemmas or stems yields improved alignment quality.

Each mapping step forms a component of the overall model. From a training point of view this means that we need to learn translation and generation tables from the word-aligned parallel corpus and define scoring methods that help us to choose between ambiguous mappings.

Phrase-based translation models are acquired from a word-aligned parallel corpus by extracting all phrase-pairs that are consistent with the word alignment. Given the set of extracted phrase pairs with counts, various **scoring functions** are estimated, such as conditional phrase translation probabilities based on relative frequency estimation or lexical translation probabilities based on the words in the phrases.

In our approach, the models for the translation steps are acquired in the same manner from a word-aligned parallel corpus. For the specified factors in the input and output, phrase mappings are extracted. The set of phrase mappings (now over factored representations) is scored based on relative counts and word-based translation probabilities.

The tables for generation steps are estimated on the output side only. The word alignment plays no role here. In fact, additional monolingual data may be used. The generation model is learned on a word-for-word basis. For instance, for a generation step that maps surface forms to part-of-speech, a table with entries such as *(fish,NN)* is constructed. One or more scoring functions may be defined over this table, in our experiments we used both conditional probability distributions, e.g., *p(fish|NN)* and *p(NN|fish)*, obtained by maximum likelihood estimation.

An important component of statistical machine translation is the language model, typically an n-gram model over surface forms of words. In the framework of factored translation models, such sequence models may be defined over any factor, or any set of factors. For factors such as part-of-speech tags, building and using higher order n-gram models (7-gram, 9-gram) is straight-forward.

Combination of Components

As in phrase-based models, factored translation models can be seen as the combination of several components (language model, reordering model, translation steps, generation steps). These components define one or more feature functions that are combined in a log-linear model:

$$\mathbf{e}|\mathbf{f}) = \exp \sum_{i=1}^{n} \lambda_i \; h_i(\mathbf{e}, \mathbf{f})$$

To compute the probability of a translation $\mathbf{e}$ given an input sentence $\mathbf{f}$, we have to evaluate each feature function h_i. For instance, the feature function for a bigram language model

component is (*m* is the number of words e_i in the sentence **e**):

$$h_{\text{lm}}(\mathbf{e}, \mathbf{f}) = p_{\text{lm}}(\mathbf{e}) = p(e_1)p(e_2|p_1)...p(e_m|e_{m-1})$$

Let us now consider the feature functions introduced by the translation and generation steps of factored translation models. The translation of the input sentence **f** into the output sentence **e** breaks down to a set of phrase translations $(\bar{f}_j, \bar{e_j})$.

For a translation step component, each feature function h_t is defined over the phrase pairs $(\bar{f}_j,\bar{e}_j)$ given a scoring function τ:

$h_t(\mathbf{e}, \mathbf{f}) = \Sigma_j \tau(\bar{f}_j, \bar{e}_j)''$

For a generation step component, each feature function h_g given a scoring function γ is defined over the output words e_k only:

$h_g(\mathbf{e}, \mathbf{f}) = \Sigma_k \gamma(e_k)$

The feature functions follow from the scoring functions (τ, γ) acquired during the training of translation and generation tables. For instance, recall our earlier example: a scoring function for a generation model component that is a conditional probability distribution between input and output factors, e.g., *γ(fish,NN,singular) = p(NN|fish)*.

The feature weights λ_i in the log-linear model are determined with the usual minimum error rate training method.

Efficient Decoding

Compared to phrase-based models, the decomposition of phrase translation into several mapping steps creates additional computational complexity. Instead of a simple table lookup to obtain the possible translations for an input phrase, now multiple tables have to be consulted and their content combined.

In phrase-based models it is easy to identify the entries in the phrase table that may be used for a specific input sentence. These are called **translation options**. We usually limit ourselves to the top 20 translation options for each input phrase.

The beam search decoding algorithm starts with an empty hypothesis. Then new hypotheses are generated by using all applicable translation options. These hypotheses are used to generate further hypotheses in the same manner, and so on, until hypotheses are created that cover the full input sentence. The highest scoring complete hypothesis indicates the best translation according to the model.

How do we adapt this algorithm for factored translation models? Since all mapping steps operate on the same segmentation, the **expansions** of these mapping steps can be efficiently pre-computed prior to the heuristic beam search, and stored as translation options. For a given input phrase, all possible translation options are thus computed before decoding (recall the earlier example, where we carried out the expansion for one input phrase). This means that the fundamental search algorithm does not change.

However, we need to be careful about combinatorial explosion of the number of translation options given a sequence of mapping steps. In other words, the expansion may create too many translation options to handle. If one or many mapping steps result in a vast increase of (intermediate) expansions, this may be become unmanageable. We currently address this problem by early pruning of expansions, and limiting the number of translation options per input phrase to a maximum number, by default 50. This is, however, not a perfect solution.

4.4 Confusion Networks Decoding

Machine translation input currently takes the form of simple sequences of words. However, there are increasing demands to integrate machine translation technology in larger information processing systems with upstream natural language and/or speech processing tools (such as named entity recognizers, automatic speech recognizers, morphological analyzers, etc.). These upstream processes tend to generate multiple, erroneous hypotheses with varying confidence. Current MT systems are designed to process only one input hypothesis, making them vulnerable to errors in the input. We extend current MT decoding methods to process multiple, ambiguous hypotheses in the form of an input lattice. A lattice representation allows an MT system to arbitrate between multiple ambiguous hypotheses from upstream processing so that the best translation can be produced.

As lattice has usually a complex topology, an approximation of it, called **confusion network**, is used instead. The extraction of a confusion network from a lattice can be performed by means of a publicly available `lattice-tool` contained in the **SRILM toolkit**. See the SRILM manual pages[4] for details and user guide.

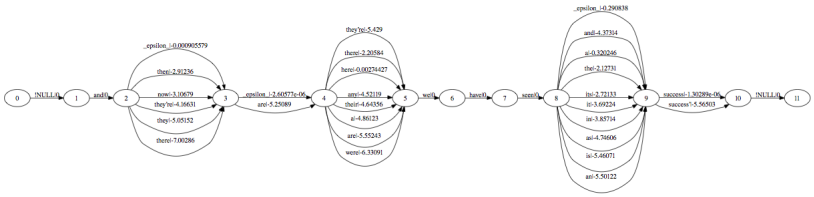
4.4.1 Confusion Networks

A **Confusion Network** (CN), also known as a **sausage**, is a weighted directed graph with the peculiarity that each path from the start node to the end node goes through all the other nodes.

Each edge is labeled with a word and a (posterior) probability. The total probability of all edges between two consecutive nodes sum up to 1. Notice that this is not a strict constraint from the point of view of the decoder; any score can be provided. A path from the start node to the end node is scored by multiplying the scores of its edges. If the previous constrain is satisfied, the product represents the likelihood of the path, and the sum of the likelihood of all paths equals to 1.

Between any two consecutive nodes, one (at most) special word `_eps_` can be inserted; `_eps_` words allows paths having different lengths.

[4] http://www.speech.sri.com/projects/srilm/manpages



Any path within a CN represents a **realization** of the CN. Realizations of a CN can differ in terms of either sequence of words or total score. It is possible that two (or more) realizations have the sae sequence of words, but different scores. Word lengths can also differ due to presence of the _eps_. This is a list of some realization of the previous CN.

```
aus der Zeitung       score=0.252       length=3
Aus der Zeitung       score=0.126       length=3
Zeitung               score=0.021       length=1
Haus Zeitungs         score=0.001       length=2
```

Notes

- A CN contains all paths of the lattice which is originated from.
- A CN can contain more paths than the lattice which is originated from (due _eps_).

4.4.2 Representation of Confusion Network

Moses adopts the following computer-friendly representation for a CN.

```
Haus 0.1 aus 0.4 _eps_ 0.3 Aus 0.2
der 0.9 _eps_ 0.1
Zeitung 0.7 _eps_ 0.2 Zeitungs 0.1
```

where a line contains the alternative edges (words and probs) between two consecutive nodes.

In the factored representation, each line gives alternatives over the full factor space:

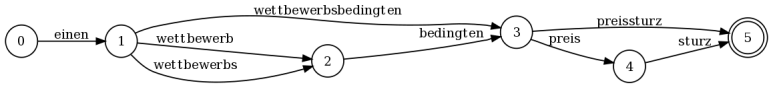
```
Haus|N 0.1 aus|PREP 0.4 Aus|N 0.4 _eps_|_eps_ 0.1
der|DET 0.1 der|PREP 0.8 _eps_|_eps_ 0.1
Zeitung|N 0.7 _eps_|_eps_ 0.2 Zeitungs|N 0.1
```

Notice that if you project the above CN on a single factor, repetitions of factors must be merged and the respective probs summed up. The corresponding word-projected CN is the one of the first example, while the part-of-speech projected CN is:

```
N 0.5 PREP 0.4 _eps_ 0.1
DET 0.1 PREP 0.8 _eps_ 0.1
N 0.8 _eps_ 0.2
```

4.5 Word Lattices

A word lattice is a directed acyclic graph with a single start point and edges labeled with a word and weight. Unlike confusion networks which additionally impose the requirement that every path must pass through every node, word lattices can represent any finite set of strings (although this generality makes word lattices slightly less space-efficient than confusion networks). However, in general a word lattice can represent an exponential number of sentences in polynomial space. Here is an example lattice showing possible ways of decompounding some compound words in German:



Moses can decode input represented as a word lattice, and, in most useful cases, do this far more efficiently than if each sentence encoded in the lattice were decoded serially. When Moses translates input encoded as a word lattice the translation it chooses maximizes the translation probability along *any* path in the input (but, to be clear, a single translation hypothesis in Moses corresponds to a single path through the input lattice).

4.5.1 How to represent lattice inputs

Lattices are encoded by ordering the nodes in a topological ordering (there may be more than one way to do this- in general, any one is as good as any other). Then, proceeding in order through the nodes, each node lists its *outgoing* edges and any weights associated with them. For example, the above lattice can be written in the moses format (also called the Python lattice format – PLF):

```
(
 (
  ('einen', 1.0, 1),
 ),
 (
  ('wettbewerbsbedingten', 0.5, 2),
  ('wettbewerbs', 0.25, 1),
  ('wettbewerb', 0.25, 1),
 ),
 (
  ('bedingten', 1.0, 1),
 ),
 (
  ('preissturz', 0.5, 2),
  ('preis', 0.5, 1),
 ),
```

```
  (
   ('sturz', 1.0, 1),
  ),
)
```

The second number is the probability associated with an edge, and the third number is the distance to the next node.

Typically, one writes lattices this with no spaces, on a single line as follows:

```
(((('einen',1.0,1),),(('wettbewerbsbedingten',0.5,2),('wettbewerbs',0.25,1), \
('wettbewerb',0.25, 1),),(('bedingten',1.0,1),),(('preissturz',0.5,2), \
('preis',0.5,1),),(('sturz',1.0,1),),)
```

4.5.2 Configuring moses to translate lattices

To indicate that moses will be reading lattices in PLF format, you need to specify `-inputtype 2` on the command line or in the moses.ini configuration file. Additionally, it is necessary to specify the feature weight that will be used to incorporate arc probability (may not necessarily be a probability!) into the translation model. To do this, add `-weight-i X` where `X` is any real number.

4.6 Publications

If you use Moses for your research, please cite the following paper in you publications:

- Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, Evan Herbst, **Moses: Open Source Toolkit for Statistical Machine Translation**, Annual Meeting of the Association for Computational Linguistics (ACL), demonstration session, Prague, Czech Republic, June 2007.

You can find out more on how Moses works from the following papers:

- Philipp Koehn and Hieu Hoang. **Factored Translation Models**, Conference on Empirical Methods in Natural Language Processing (EMNLP), Prague, Czech Republic, June 2007.
- Richard Zens and Hermann Ney. **Efficient Phrase-table Representation for Machine Translation with Applications to Online MT and Speech Translation**, Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL), Rochester, NY, April 2007.
- Nicola Bertoldi, Richard Zens, Marcello Federico and Wade Shen, **Efficient Speech Translation through Confusion Network Decoding**, IEEE Transactions on Audio, Speech, and Language Processing, vol. 16, no. 9, pp. 1696-1705, 2008

5

Code Guide

5.1 Code Guide

This section gives a overview of the code. All the source code is commented for Doxygen, so you can browse a current snapshot of the source documentation[1]. Moses is implemented using object-oriented principles, and you can get a good idea of its class organization from this documentation

The source code is in two directories:

- `moses-cmd/src` contains code relevant to the command line version of the decoder (currently that is the only one, a client-server version is planned)
- `moses/src` contains the bulk of the code

In the following, we provide a short walk-through of the decoder.

5.1.1 Quick Start

- The main function: `moses-cmd/src/Main.cpp`
- Initialize the decoder
  - `moses/src/Parameter.cpp` specifies parameters
  - `moses/src/StaticData.cpp` contains globals, loads tables
- Process a sentence
  - `Manager.cpp` implements the decoding algorithm
  - `TranslationOptionCollection.cpp` contains translation options
  - `Hypothesis.cpp` represents partial translation

[1] http://www.statmt.org/moses/html/hierarchy.html

  - `HypothesisStack.cpp` contain viable hypotheses, implements pruning

- Output results: `moses-cmd/src/Main.cpp`
  - `moses-cmd/src/IOStream::OutputBestHypo` print best translation
  - n-best lists generated in `Manager.cpp`, output in `IOStream.cpp`

5.2 Coding Style

To ensure maintainability and consistency, please follow the recommendations below when developing Moses.

- Code for cross-platform compatibility.
- Tabs should be displayed as 2 spaces. However, donŠt convert tabs to spaces
- Putting opening braces on a separate line, e.g.

```
if (expr) // correct
{
  ...
}

if (expr) { // wrong
  ...
}
```

- Start all functions and class names with capital letters. Start all variable with small letter. Start all class variable with `m_`. For example

```
void CalcNBest(size_t count, LatticePathList \&ret) const;
Sentence m_source;
```

- Do not use Hungarian notations.
- Use object-orientated designs, including
  - Create Get/Set functions rather than exposing class variables.
  - Label functions, variables and arguments as const where possible.
  - Prefer references over pointers
  - General styles
  - Prefer enum types over integers
  - Resolve compiler warnings as well as errors

- Delete tracing/debugging code once they are not needed.

Source Control Etiquette

- Do not check in non-compilable code, or if functionality is reduced
- Ignore the above if you need to, just let people know
- Check-in your work often to avoid resolution conflicts
- Add log messages to check-ins
- Check in make/project files. However, you are not required to update project files other than the ones you use.

5.3 Factors, Words, Phrases

Moses is the implemented of a factored translation model. This means that each word is represented by a vector of factors, which are typically word, part-of-speech tags, etc. It also means that the implementation is a bit more complicated than a non-factored translation model.

This section intends to provide some documentation of how factors, words, and phrases are implemented in Moses.

5.3.1 Factors

The class `Factor`[2] implements the most basic unit of representing text in Moses. In essence it is a string.

Factors do not know about their own type (which component in the word vector they represent), this is referred to as its `FactorType` when needed. This factor type is implemented as a `size_t`, i.e. an integer. What a factor really represents (be it a surface form or a part of speech tag), does not concern the decoder at all. All the decoder knows is that there are a number of factors that are referred to by their factor type, i.e. an integer index.

Since we do not want to store the same strings over and over again, the class `FactorCollection`[3] contains all known factors. The class has one global instance, and it provides the essential functions to check if a newly constructed factor already exists and to add a factor. This enables the comparison of factors by the cheaper comparison of the pointers to factors. Think of the `FactorCollection` as the global factor dictionary.

[2] http://www.statmt.org/moses/html/classMoses_1_1Factor.html
[3] http://www.statmt.org/moses/html/classMoses_1_1FactorCollection.html

5.3.2 Words

A word is, as we said, a vector of factors. The class `Word`[4] implements this. As data structure, it is a array over pointers to factors. This does require the code to know what the array size is, which is set by the global `MAX_NUM_FACTORS`. The word class implements a number of functions for comparing and copying words, and the addressing of individual factors.

Again, a word does not know, how many factors it really has. So, for instance, when you want to print out a word with all its factors, you need to provide also the factor types that are valid within the word. See the function `Word::GetString` for details.

5.3.3 Factor Types

This is a good place to note that referring to words gets a bit more complicated. If more than one factor is used, it does not mean that all the words in the models have all the factors. Take again the example of a two-factored representation of words as surface form and part-of-speech. We may still use a simple surface word language model, so for that language model, a word only has one factor.

We expect the input to the decoder to have all factors specified and during decoding the output will have all factors of all words set. The process may not be a straight-forward mapping of the input word to the output word, but it may be decomposed into several mapping steps that either translate input factors into output factors, or generate additional output factors from existing output factors.

At this point, keep on mind that a `Factor` has a `FactorType` and a `Word` has a `vector<FactorType>`, but these are not internally stored with the `Factor` and the `Word`.

Related to factor types is the class `FactorMask`[5], which is a bit array indicating which factors are valid for a particular word.

5.3.4 Phrases

Since decoding proceeds in the translation of input phrases to output phrases, a lot of operation involve the class `Phrase`[6]. Phrases know a little bit more about their role in the world. For instance, they know if they are `Input` or `Output` phrases, which they reveal with a function call to `Phrase::GetDirection()`. Since the total number of input and output factors is known to the decoder (it has to be specified in the configuration file `moses.ini`), phrases are also a bit smarter about copying and comparing.

The `Phrase` class implements many useful functions, and two other classes are derived from it:

[4] http://www.statmt.org/moses/html/classMoses_1_1Word.html
[5] http://www.statmt.org/moses/html/classMoses_1_1FactorMask.html
[6] http://www.statmt.org/moses/html/classMoses_1_1Phrase.html

- The simplest form of input, a sentence as string of words, is implemented in the class `Sentence`[7].
- The class `TargetPhrase`[8] may be somewhat misleadingly named, since it not only contains a output phrase, but also a phrase translation score, future cost estimate, pointer to source phrase, and potentially word alignment information.

5.4 Adding Feature Functions

(note: thanks to Jason Katz-Brown for a earlier version)

The log-linear model underlying statistical machine translation allows for the combination of several components that each weigh in on the quality of the translation. Each component is represented by one or more features, which are weighted, and multiplied together.

Formally, the probability of a translation **e** of an input sentence **f** is computed as

$$p(\mathbf{e}|\mathbf{f}) = \prod_i h_i(\mathbf{e},\mathbf{f})^{\lambda_i} \tag{5.1}$$

where h_i are the feature functions and λ_i the corresponding weights.

Note that the decoder internally uses logs, so in fact what is computed is

$$\log p(\mathbf{e}|\mathbf{f}) = \sum_i \log(h_i(\mathbf{e},\mathbf{f}))\ \lambda_i \tag{5.2}$$

The tuning (page 98) stage of the decoder is used to set the weights.

The following components are typically used:

- phrase translation model (5 features, described here (page 85))
- language model (1 feature)
- distance-based reordering model (1 feature)
- word penalty (1 feature)
- lexicalized reordering model (6 features, described here (page 87))

One way to attempt to improve the performance of the system is to add additional feature functions. This section explains what needs to be done to add a feature. Unless otherwise specified the Moses code files are in the directory `moses/src`. In the following we refer to the new feature as `xxx`.

[7] http://www.statmt.org/moses/html/classMoses_1_1Sentence.html
[8] http://www.statmt.org/moses/html/classMoses_1_1TargetPhrase.html

Side note: Adding a new component may imply that several scores are added. In the following, as in the Moses source code, we refer to both components and scores as *features*. So, a feature may have multiple features. Sorry about the confusion.

5.4.1 Score Producer

The feature computes one or more values, and we need to write a **score producer** for it.

One important question about the new feature is, if it depends on just on the current phrase translation, or also on prior translation decision. We call the first case **stateless**, the second **stateful**. The second case causes additional complications for the dynamic programming strategy of recombining hypotheses. If two hypotheses differ in their past translation decisions which matters for the new feature, then they cannot be recombined.

For instance, the word penalty does only depend on the current phrase translation and is hence stateless. The distortion features also depend on the previous phrase translation and they are hence stateful.

For some simple feature functions, see `DummyScoreProducers.h` and `DummyScoreProducers.cpp`, but typically a new feature is more evolved, for instance, it requires reading in a model file and representing it with a data structure and more complex computations. See `LexicalReordering.h` and `LexicalReordering.cpp` for such a more complex feature.

In the following, we assume such a more complex feature, which is implemented in its own source files `XXX.h` and `XXX.cpp`. The feature is implemented as a class which inherits from either `StatefulFeatureFunction` or `StatelessFeatureFunction`. So, you will write some code in `XXX.h` that starts with

```
namespace Moses
{
class XXX : public StatefulFeatureFunction {
  ...
}
```

The constructor has to contain the two lines below to inform the decoder about the score producer:

```
const_cast<ScoreIndexManager\&> \
  (StaticData::Instance().GetScoreIndexManager()) \
  .AddScoreProducer(this);
const_cast<StaticData\&>(StaticData::Instance()) \
  .SetWeightsForScoreProducer(this, weights);
```

This class has to include the public functions

- `std::string GetScoreProducerDescription() const;` This function returns the name of the feature (by convention without spaces).

- `std::string GetScoreProducerWeightShortName() const;` This function returns the short name of the weight for this feature.
- `size_t GetNumScoreComponents() const;` This function returns returns the number of features that the component uses. See some examples above, often this is 1.

If it is a stateless feature, then the class has to contain the functions

- `void Evaluate( const TargetPhrase&, ScoreComponentCollection* ) const;` This function calculates the feature value for a new hypothesis that was created and returns it in a vector of floats.

If it is a stateful feature, then the class has to contain the functions

- `void Evaluate( const Hypothesis&, const FFState*, ScoreComponentCollection* ) const;` This function calculates the feature value for a new hypothesis that was created and returns it in a vector of floats.
- `const FFState* EmptyHypothesisState() const;`

If you are programming with a rapid prototyping strategy (which we would recommend), you should first implement some dummy functions here, complete the integration work of the feature into the decoder (see below), test that, and then implement the nuts and bolts of the feature.

For instance, you may implement the following skeletons for a stateful feature function in `XXX.cpp`:

```
std::string XXX::GetScoreProducerDescription() const
{
  return "XXXFeature";
}

std::string XXX::GetScoreProducerWeightShortName() const
{
  return "x";
}

size_t XXX::GetNumScoreComponents() const
{
  return 1;
}
```

For a stateful feature function, you also need to implement:

```
FFState* XXX::Evaluate( const Hypothesis\& cur_hypo,
                        const FFState* prev_state,
                        ScoreComponentCollection* accumulator) const {
  accumulator->PlusEquals( this, -1.0 );
  return NULL;
}

const FFState* EmptyHypothesisState() const
{
  return NULL;
}
```

For a stateless feature function, only the function `Evaluate` needs to be implemented, which takes different parameters (see above), but otherwise functions the same.

5.4.2 Adding Decoder Parameters

The feature requires a weight and typically also loading of data files (e.g., the language model, the phrase table, or the reordering table). These settings need to be communicated to the decoder.

The file `Parameter.cpp` contains the function `Parameter::Parameter()` where you can specify each parameter. You have to add a line for each parameter, for instance:

```
AddParam("weight-xxx", "x", "weight for xxx feature");
AddParam("xxx-file", "xf", "model file for xxx");
```

The first value of the function `AddParam` is the full name of the feature, the second the short name, and the third the help message.

5.4.3 Integrating the Feature

In `StaticData.cpp`, some code needs to be added to process the weight parameter and to create an instance of the score producer. The function `StaticData::LoadData` is called upon start-up of the decoder and carries out all the necessary initializations.

On top of the file, you need to add a line

```
#include "XXX.h"
```

You also need to write a function function `LoadXXX()` which is also placed in the file `StaticData.cpp` and called within `StaticData::LoadData(Parameter *parameter)` (see for the other loading functions for the proper placement).

The loading function has to intialize the weights and load any data files, if such exist. Typically, this looks something like this:

```
bool StaticData::LoadXXX()
{
  const vector<float> \&weight = Scan<float>(m_parameter->GetParam("weight-xxx"));
  const vector<string> \&file = m_parameter->GetParam("xxx-file");

  if (weight.size() != file.size())
  {
    // mismatch in number of weights and file, write some error message
    return false;
  }

  if (weight.size() == 1) // check if feature is used
  {
    m_XXX = new XXX(weight[0], file[0]); // create the feature
    if (!m_XXX) return false;
  }
  return true;
}
```

Your feature may have a different number of weights, use a different number of files, and have additional parameters. There may also be multiple instances of the features for different factors (or other variations). In short: Your mileage may vary.

The feature instance(s) `m_XXX` have to be deleted in `StaticData:: StaticData`:

```
delete m_XXX; // if only one instance is possible
RemoveAllInColl(m_XXX); // if multiple instances are possible
```

In `StaticData.h`, you need to specify globals such as `m_XXX`:

```
XXX* m_XXX; // if only one instance is possible
std::vector<XXX*> m_XXX; // if multiple instances are possible
```

`StaticData.h` must also include a declaration of the class you are using.

```
class XXX;
```

At this point, it is a good idea if all this compiles. If it does, you can already run the decoder with the new feature function. Even if it does not do anything useful, you can check if the feature function is properly called. For instance, when you generate an n-best list along with the output, you can see if the feature function was properly included in the scoring.

5.4.4 Implementing the Feature

From here on, it should be pretty straight-forward to implement the feature, since all the integration work has been done. Nevertheless, here some comments on how to handle certain non-trivial issues.

Loading Models

Most features will involve a model that is stored in a data file. Typically this data file is loaded at start-up time. You should place the code to load the data file in the constructor function.

Alternatively, the model may remain on disk or on a remote server. In this case, it may be useful to do some pre-computation before decoding each sentence and/or some caching.

Initialization for a Sentence

The function `StaticData::InitializeBeforeSentenceProcessing(InputType const& in)` in `StaticData.cpp` is called before each sentence is translated.

If your feature requires pre-computations, then here is a good place to place a call to a function of your feature class that implements the pre-computation. This function is conventionally called `InitializeForInput( InputType const& in )`.

If the pre-computation requires that the translation options are already collected (also a form of pre-computation), then you need to place a call to your initialization function after the line

```
m_transOptColl->CreateTranslationOptions(decodeStepVL);
```

in `Manager::ProcessSentence()` in `Manager.cpp`. In this case, you need to make sure that the generated precomputated data structure is destroyed in the destructor function. An instance for a Manager object is created for each sentence.

Caching across Sentences

In unusual cases, you may also do some caching across sentences. This is done, for instance, for the translation options that are retrieved from the on-disk binary file on the fly and that require expensive computations in some cases. This cache should be implemented within your feature function class, but you should also make sure that the cache is frequently cleared.

5.4.5 Tuning

Minimum error train training (MERT) needs to know about the new feature. You have to adjust the script `mert-moses.pl` in a few places. First, the script needs to know about your feature, so you have to had it to the following list, which maps the full short names of the features to the full names. In our example, we would add `x=weight-xxx`:

```
my \$ABBR_FULL_MAP =
 "d=weight-d lm=weight-l tm=weight-t w=weight-w g=weight-generation lex=weight-lex";
```

If the feature is always present in decoding and does not depend on an optional model file, you need to add its range of possible weights to `$default_triples`. This range is specified as

triples (start, min, max). If the feature may have multiple weights, you have to define multiple triples. This should look something like this:

```
"x"  => [ [ 1.0, 0.0, 2.0 ] ], ### xxx feature
```

In the case that the feature depends on the inclusion of a model, the script checks the configuration file for weights that need to be set. You also have to include triplets, but this time to the hash `$additional_triples`. You also have to help MERT out in finding out if the feature is used. You have to add the model file that triggers your feature to the following line, e.g, by including `xxx-file=x`.

```
my \$TABLECONFIG_ABBR_MAP =
  "ttable-file=tm lmodel-file=lm distortion-file=d generation-file=g";
```

Since the specification of a model includes information such as the file name and possibly the number of features, you need to tell MERT where to find this information in `%where_is_filename` and `%where_is_lambda_count`.

5.5 Regression Testing

5.5.1 Goals

The goal of regression testing is to ensure that any changes made to the decoder do not break what has been determined to be correct, previously. The regression test suite is fast enough to run often, but still should provide adequate confidence that nothing substantial has changed about the internal workings of moses. The regression test suite is designed to run on most UNIX-like systems.

5.5.2 Test suite

The following regression tests are currently implemented:

- `basic-surface-only` Tests basic translation, compares output strings and probabality scores.
- `basic-surface-binptable` Tests binary phrase table
- `ptable-filtering` Tests the filtering of the phrase table by estimated phrase cost, ensures that the estimated phrase cost stays the same and that the same list of phrases is consistent. Matches pharaoh.
- `multi-factor` Test that moses can do translation with two factors (Currently does a very basic test- it should be enhanced to at least include OOV words).

- `multi-factor-binptable` Tests factored setup with binary phrase table.
- `multi-factor-drop` Test of dropping words in a multi-factor model.
- `nbest-multi-factor` Tests n-best list generation for multi-factor models
- `n-best` Test n-best filtering, ensure consistency of top scores and score components. This will require ensuring that any moses binary is capable of generating n-best lists.
- `lattice-surface` Tests lattice decoding
- `lattice-distortion` Tests lattice decoding with distortion (?)
- `confusionNet-surface-only` Tests confusion network decoding
- `confusionNet-multi-factor` Tests confusion network decoding with multiple factors
- `lexicalized-reordering` Tests lexical reordering model
- `lexicalized-reordering-cn` Tests lexical reordering model in combination with confusion network
- `xml-markup` Tests XML Markup in input to specify translations

5.5.3 Running the test suite

Running the regression test suite requires that you have checked out the regression-testing module from SVN (see SVN Instructions[9], if you downloaded the entire trunk you have this automatically).

You will also need the data file from http://www.statmt.org/moses/reg-testing/moses-reg-test-data-2.tgz

Once you have all that, running is as easy as:

```
/<MYCVS-ROOT>/regression-testing/run-test-suite --decoder <PATH/TO/moses> \
  --data-dir PATH/TO/moses-reg-test-data-2
```

If all goes well, you will see a list of the tests run, their status (hopefully pass), and a path where the results are archived.

5.5.4 How it works

The test suite invokes moses to decode a few sample phrases with well-known models. The output from these invocations is then scraped for information (for example, the output translation of a sentence or its probability score) which is stored in a file called `results.dat`. These

[9] http://sourceforge.net/svn/?group_id=171520

values are then compared to a ground truth, which was established either by hand, from a prior moses run, or from a pharaoh run.

Each test in the test suite can be run individually. To run the phrase table filtering test (called ptable-filtering) do the following:

```
<MYCVS-ROOT>/regression-testing/run-single-test.pl --decoder <PATH/TO/moses> \
  --data-dir PATH/TO/moses-reg-test-data-2 --test ptable-filtering
```

This will provide a point-by-point analysis of each failure or success in the test as well as information.

Note: Since the test suite relies on the output of moses, changes to the output format may result in broken tests. If you make changes that affect presentation only, you will need to update the testing filters (which convert the raw moses output into the `results.dat` format).

5.5.5 Writing regression tests

Writing regression tests is easy, but since these tests must be able to be run anywhere, it is important to keep a few things in mind. First, check out the regression-testing module from SVN. Settle on what you would like to test in and choose a test name (henceforth, this name will be `TEST-NAME`). Create a directory for it under regression testing.

Place the following into the directory `regression-testing/tests/TEST-NAME`:

- `to-translate`, which contains the text that will be translated by moses.
- `moses.ini`. This moses.ini file should have *no absolute paths*. All paths should be expressed in terms of the variables `${LM_PATH}` and `${MODELS_PATH}`.
- The filter files, `filter-stderr` and `filter-stdout`. These files should read from `STDIN` and write results of the form `KEY = value` to `STDOUT`. No other output should be generated. Numeric values (such as times) that do not require exact matches can have the form `KEY value`. These files are the trickiest part about writing a new regression test. However, they allow great flexibility in verifying specific aspects of a decoding run.
- `truth/results.dat` This file should have the values (as produced by filter-stderr and filter-stdout) that are expected from the test run.

If you need to add language models, phrase tables, generation tables or anything like this, you will need to increment the required data version number in `MosesRegressionTesting.pm`. Then, you will need to create a new `.tgz` file that contains the data for all the tests (the data dependencies are not checked into SVN because they are extremely large). This must then be made available for download.

6

Reference

6.1 Frequently Asked Questions

6.1.1 Content

• My system is taking a really long time to translate a sentence. What can I do to speed it up ? (page 144)

• The system runs out of memory during decoding. (page 144)

• I would like to point out a bug / contribute code. (page 144)

• How can I get an updated version of Moses ? (page 144)

• When is version 1/beta of Moses coming out? (page 145)

• I'm an undergrad/masters student looking for a project in SMT. What should I do? (page 145)

• What do the 5 numbers in the phrase table mean? (page 145)

• What OS does Moses run on? (page 145)

• Can I use Moses on Windows ? (page 146)

• Do I need a computer cluster to run experiments? (page 146)

• I've compiled Moses, but it segfaults when running. (page 146)

• How do I add a new feature function to the decoder? (page 146)

• Compiling with SRILM or IRSTLM produces errors. (page 146)

• I'm trying to use Moses to create a web page to do translation. (page 147)

• How can a create a system that translate both ways, ie. X-to-Y as well as Y-to-X ? (page 147)

• PhraseScore dies with signal 11 - why? (page 147)

• Does Moses do Hierarchical decoding, like Hiero etc? (page 148)

• Can I use Moses in proprietary software ? (page 148)

• Who do I ask if my question hasn't been answered by this FAQ? (page 148)

6.1.2 My system is taking a really long time to translate a sentence. What can I do to speed it up ?

The single best thing you can do is to binarize the phrase tables and language models. See question below also.

6.1.3 The system runs out of memory during decoding.

Filter and binarize your phrase tables. Binarize your language models using the IRSTLM. Binarize your lexicalized re-ordering table.

Binarizing the phrase table helps decrease memory usage as only phrase pairs that are needed for each sentence are read from file into memory. Similarly for language models and lexicalized reordering models.

This webpage (page 51) tell you how to binarize the models.

6.1.4 I would like to point out a bug / contribute code.

We are always grateful for bug reports and code contribution. Send it to an existing Moses developer you work with, or send it to Hieu Hoang at Edinburgh University.

If you want to check it code yourself, create a Sourceforge account here[1]

Then ask one of the project admins to add you to the Moses project. The admins are currently

- Chris Callison-Burch
- Hieu Hoang
- Nicola Bertoldi
- Philipp Koehn
- Chris Dyer

We'll prob ask to code review you a few times before giving you free reign. However, there is less oversight if you intend to work on your own branch, rather than the trunk.

6.1.5 How can I get an updated version of Moses ?

The best way is using SVN.

From the command line, type

```
svn up
```

Or use whatever GUI client you have.

[1] https://sourceforge.net/account/registration/

6.1.6 When is version 1/beta of Moses coming out?

Moses is an academic project. There's no usability/reliability testing, so there'll probably never be a version 1.

However, there is some regression testing, and no-one would be able to use it if it weren't reliable. And it's being continually updated with development in the community so you can be sure you'll get good results.

We can't give you a stronger guarantee than that.

6.1.7 I'm an undergrad/masters student looking for a project in SMT. What should I do?

Email the mailing list with the title: 'Code monkey available. Will work for peanuts' ! Seriously, there's lots and lots of projects available. There has been 3-4 months projects in the past which have made a significant contribution to the community and have been integrated into the Moses toolkit. Your contribution will be grateful appreciated. Talk to your professor in the 1st instance, then talk to us.

6.1.8 What do the 5 numbers in the phrase table mean?

See the section on advanced features (page 51)

6.1.9 What OS does Moses run on?

It depends on which part.

The decoder can be compiled and run on Linux (32 and 64-bits), Windows, cygwin, Mac OSX (Intel and PowerPC). Unconfirmed reports of the decoder running on Solaris and BSD too.

The training and tuning scripts can only reliably run on Linux (32 and 64-bits). Issues that prevents these scripts running on other platforms include:

- File system case-sensitivity
- `zcat`, `gzip` command line programs missing
- GIZA++ only compilable by specific gcc versions
- Availability of Sun Grid Engine

Therefore, the only realistic OS to run the whole SMT pipeline on is Linux.

6.1.10 Can I use Moses on Windows ?

Yes, but you must use cygwin. The training & tuning scripts, and external LM libraries were developed for Unix-like OS's so it is difficult to run without the cygwin environment.

SRI and IRST language models are available for Unix-like platforms, not Windows. The only language model available on Windows is an internal LM, which is slow and memory hungry.

The training & tuning scripts are written in Perl/Python/gcc-specific C++ which will be difficult to compile or run without a Unix environment.

However, the decoder is cross-platform and can be compiled and developed with Visual Studio. The VS 2005 project files are included.

6.1.11 Do I need a computer cluster to run experiments?

The Moses toolkit uses SGE (Sun Grid Engine) cluster to parallelize tasks. Even though it is not strictly necessary to use a cluster to run your experiments, it is highly advisable to get your experiments to run faster.

The most CPU intensive task is the tuning of the weights (MERT tuning). As an indication, a Europarl trained model, using 2000 sentences for tuning, takes 1-2 days to tune using 15 CPUs. 10-15 iterations are typical.

6.1.12 I've compiled Moses, but it segfaults when running.

First of all, try to identify the fault yourself. The most common error is the ini file isn't correct, or the sentence input is badly formatted.

If necessary, you can debug the system by stepping through the source code. We put a lot of effort into making the code easy to read and debug. Also, the decoder comes with Visual Studio, XCode, and Eclipse project file to help you debug in a GUI environment.

If you still can't find the solution, email the mailing list. Its useful to attach the ini file, the output just before it crashes, and any other info that you think may be useful to help resolve the problem.

6.1.13 How do I add a new feature function to the decoder?

This is now documented in its own section (page 133).

6.1.14 Compiling with SRILM or IRSTLM produces errors.

Firstly, make sure SRILM/IRSTLM themselves have compiled successfully. You should see be a libflm.a/libdstruct.a etc (for SRILM), or libirstlm.a. If these aren't available, then something

went wrong. SRILM and IRSTLM are external libraries so the Moses developers have limited say and knowledge of them.

If Moses still doesn't compile successfully, look at the compile error to see where the compiler is trying to find these external libraries. Occasionally (especially when compiling on 64-bit machines), Moses expects the .a file in 1 subdirectory but they are in another. This is easily solved by moving copying the .a file to the place where Moses expect it to be.

6.1.15 I'm trying to use Moses to create a web page to do translation.

There's a subproject in moses, /web , which allows you to set up a web page to translate other web pages. Its written in perl and the installation is non-trivial. Follow the instructions carefully.

It doesn't translate ad-hoc sentences. If you have some code which allow translation of ad-hoc sentences, please share it with us !

6.1.16 How can a create a system that translate both ways, ie. X-to-Y as well as Y-to-X ?

You need to do everything twice, and run 2 decoders. There is a lot of overlap between them, but the toolkit is designed to go 1 way at a time.

6.1.17 PhraseScore dies with signal 11 - why?

This may happen means because you have a null byte in your data. Look at line 2 of model/lex.f2e.

Try this to find lines with null bytes in your original data:

```
grep -Pc '[\000]' <files ...>
```

(If your grep doesn't support Perl-style regepx syntax (`-P`), you'll have to express that a different way.)

If this turns out to be the problem, and you don't want to run GIZA again from scratch, you can try the following:

First go into `working-dir/model` and delete everything but the following:

```
aligned.grow-diag-final-and
aligned.0.fr
aligned.0.en
lex.0-0.n2f
lex.0-0.f2n
```

Now run this fragment of Perl:

```
perl -i.BAD -pe 's/[\000]/NULLBYTE/g;' aligned.0* lex.0*
```

This will replace every null byte in those four files, saving the old version out to `*.BAD`. (This may be overkill, for instance if only the foreign side has the problem.

Now restart the moses training script with the same invocation as before, but tell it to start at step 5:

```
train-factored-phrase-model.perl ... --first-step 5
```

6.1.18 Does Moses do Hierarchical decoding, like Hiero etc?

Not yet but we're working on it. The Work-In-Progress is in a branch on sourceforge: `/branches/mt3_chart`. You are welcome to play with it and contribute. Email Hieu Hoang for more info.

6.1.19 Can I use Moses in proprietary software ?

Moses is licensed under the LGPL. Google for this term to see its implication.

Basically, if you're just using moses unchanged, yes. If you make changes to the code, you have to share it with everyone.

6.1.20 Who do I ask if my question hasn't been answered by this FAQ?

Send questions to the mailing list 'moses-support'. However, you have to sign up[2] before emailing.

6.2 Reference: All Decoder Parameters

- `-beam-threshold` (`b`): threshold for threshold pruning
- `-cache-path`: ?
- `-config` (`-f`): location of the configuration file
- `-constraint`: Target sentence to produce
- `-cube-pruning-diversity` (`-cbd`): How many hypotheses should be created for each coverage. (default = 0)
- `-cube-pruning-pop-limit` (`-cbp`): How many hypotheses should be popped for each stack. (default = 1000)
- `-distortion`: configurations for each factorized/lexicalized reordering model.
- `-distortion-file`: source factors (0 if table independent of source), target factors, location of the factorized/lexicalized reordering tables

[2] http://mailman.mit.edu/mailman/listinfo/moses-support

- `-distortion-limit` (`-dl`): distortion (reordering) limit in maximum number of words (0 = monotone, -1 = unlimited)
- `-drop-unknown` (`-du`): drop unknown words instead of copying them
- `-early-discarding-threshold` (-edt@@): threshold for constructing hypotheses based on estimate cost
- `-factor-delimiter` (`-fd`): specify a different factor delimiter than the default
- `-generation-file`: location and properties of the generation table
- `-include-alignment-in-n-best`: include word alignment in the n-best list. default is false
- `-input-factors`: list of factors in the input
- `-input-file` (`-i`): location of the input file to be translated
- `-inputtype`: text (0), confusion network (1) or word lattice (2)
- `-labeled-n-best-list`: print out labels for each weight type in n-best list. default is true
- `-lmodel-dub`: dictionary upper bounds of language models
- `-lmodel-file`: location and properties of the language models
- `-lmstats` (`-L`): (1/0) compute LM backoff statistics for each translation hypothesis
- `-mapping`: description of decoding steps
- `-max-partial-trans-opt`: maximum number of partial translation options per input span (during mapping steps)
- `-max-phrase-length`: maximum phrase length (default 20)
- `-max-trans-opt-per-coverage`: maximum number of translation options per input span (after applying mapping steps)
- `-mbr-scale`: scaling factor to convert log linear score probability in MBR decoding (default 1.0)
- `-mbr-size`: number of translation candidates considered in MBR decoding (default 200)
- `-minimum-bayes-risk` (`-mbr`): use miminum Bayes risk to determine best translation
- `-monotone-at-punctuation` (`-mp`): do not reorder over punctuation
- `-n-best-factor`: factor to compute the maximum number of contenders (=factor*nbest-size). value 0 means infinity, i.e. no threshold. default is 0

- `-n-best-list`: file and size of n-best-list to be generated; specify - as the file in order to write to STDOUT
- `-output-factors`: list if factors in the output
- `-output-search-graph` (`-osg`): Output connected hypotheses of search into specified filename
- `-output-word-graph` (`-owg`): Output stack info as word graph. Takes filename, 0=only hypos in stack, 1=stack + nbest hypos
- `-persistent-cache-size`: maximum size of cache for translation options (default 10,000 input phrases)
- `-phrase-drop-allowed` (`-da`): if present, allow dropping of source words
- `-print-alignment-info`: Output word-to-word alignment into the log file. Word-to-word alignments are taken from the phrase table if any. Default is false
- `-print-alignment-info-in-n-best`: Include word-to-word alignment in the n-best list. Word-to-word alignments are takne from the phrase table if any. Default is false
- `-recover-input-path` (`-r`): (confusion net/word lattice only) - recover input path corresponding to the best translation
- `-report-all-factors`: report all factors in output, not just first
- `-report-segmentation` (`-t`): report phrase segmentation in the output
- `-search-algorithm`: Which search algorithm to use. 0=normal stack, 1=cube pruning (default = 0)
- `-stack` (`-s`): maximum stack size for histogram pruning
- `-stack-diversity` (`-sd`): minimum number of hypothesis of each coverage in stack (default 0)
- `-time-out`: seconds after which is interrupted (-1=no time-out, default is -1)
- `-translation-details` (`-T`): for each best translation hypothesis, print out details about what source spans were used, dropped
- `-translation-option-threshold` (`-tot`): threshold for translation options relative to best for input phrase
- `-ttable-file`: location and properties of the translation tables
- `-ttable-limit` (`-ttl`): maximum number of translation table entries per input phrase

- `-use-alignment-info`: Use word-to-word alignment: actually it is only used to output the word-to-word alignment. Word-to-word alignments are taken from the phrase table if any. Default is false.
- `-use-persistent-cache`: cache translation options across sentences (default true)
- `-verbose` (`-v`): verbosity level of the logging
- `-weight-d` (`-d`): weight(s) for distortion (reordering components)
- `-weight-e` (`-e`): weight for word deletion
- `-weight-file` (`-wf`): file containing labeled weights
- `-weight-generation` (`-g`): weight(s) for generation components
- `-weight-i` (`-I`): weight for word insertion
- `-weight-l` (`-lm`): weight(s) for language models
- `-weight-t` (`-tm`): weights for translation model components
- `-weight-w` (`-w`): weight for word penalty
- `-xml-input` (`-xi`): allows markup of input with desired translations and probabilities. values can be 'pass-through' (default), 'inclusive', 'exclusive', 'ignore'

6.3 Reference: All Training Parameters

- `-root-dir` – root directory, where output files are stored
- `-corpus` – corpus file name (full pathname), excluding extension
- `-e` – extension of the English corpus file
- `-f` – extension of the foreign corpus file
- `-lm` – language model: <factor>:<order>:<filename> (option can be repeated)
- `-first-step` – first step in the training process (default 1)
- `-last-step` – last step in the training process (default 7)
- `-parts` – break up corpus in smaller parts before GIZA++ training
- `-corpus-dir` – corpus directory (default `$ROOT/corpus`)
- `-lexical-dir` – lexical translation probability directory (default `$ROOT/model`)

- `-model-dir` – model directory (default `$ROOT/model`)
- `-extract-file` – extraction file (default `$ROOT/model/extract`)
- `-giza-f2e` – GIZA++ directory (default `$ROOT/giza.$F-$E`)
- `-giza-e2f` – inverse GIZA++ directory (default `$ROOT/giza.$E-$F`)
- `-alignment` – heuristic used for word alignment: intersect, union, grow, grow-final, grow-diag, grow-diag-final (default)
- `-max-phrase-length` – maximum length of phrases entered into phrase table (default 7)
- `-giza-option` – additional options for GIZA++ training
- `-verbose` – prints additional word alignment information
- `-no-lexical-weighting` – only use conditional probabilities for the phrase table, not lexical weighting
- `-parts` – prepare data for GIZA++ by running `snt2cooc` in parts
- `-direction` – run training step 2 only in direction 1 or 2 (for parallelization)
- `-reordering` –
- `-reordering-smooth` –
- `-alignment-factors` –
- `-translation-factors` –
- `-reordering-factors` –
- `-generation-factors` –
- `-decoding-steps` –

6.3.1 Basic Options

A number of parameters are required to point the training script to the correct training data. We will describe them in this section. Other options allow for partial training runs and alternative settings.

As mentioned before, you want to create a special directory for training. The path to that directory has to be specified with the parameter `-root-dir`.

The root directory has to contain a sub directory (called `corpus`) that contains the training data. The training data is a parallel corpus, stored in two files, one for the English sentences,

one for the foreign sentences. The corpus has to be sentence-aligned, meaning that the 1624th line in the English file is the translation of the 1624th line in the foreign file.

Typically, the data is lowercased, no empty lines are allowed, and having multiple spaces between words may cause problems. Also, sentence length is limited to 100 words per sentence. The sentence length ratio for a sentence pair can be at most 9 (i.e, having a 10-word sentence aligned to a 1-word sentence is disallowed). These restrictions on sentence length are caused by GIZA++ and may be changed (see below).

The two corpus files have a common file stem (say, `euro`) and extensions indicating the language (say, `en` and `de`). The file stem (`-corpus-file`), and the language extensions (`-e` and `-f`) have to be specified to the training script.

In summary, the training script may be invoked as follows:

```
train-phrase-model.perl --root-dir . --f de --e en --corpus corpus/euro >\& LOG
```

After training, typically the following files can be found in the root directory (note the time stamps that tell you something about how much time was spent on each step took for this data):

```
> ls -lh *
-rw-rw-r--    1 koehn    user         110K Jul 13 21:49 LOG

corpus:
total 399M
-rw-rw-r--    1 koehn    user         104M Jul 12 19:58 de-en-int-train.snt
-rw-rw-r--    1 koehn    user         4.2M Jul 12 19:56 de.vcb
-rw-rw-r--    1 koehn    user         3.2M Jul 12 19:42 de.vcb.classes
-rw-rw-r--    1 koehn    user         2.6M Jul 12 19:42 de.vcb.classes.cats
-rw-rw-r--    1 koehn    user         104M Jul 12 19:59 en-de-int-train.snt
-rw-rw-r--    1 koehn    user         1.1M Jul 12 19:56 en.vcb
-rw-rw-r--    1 koehn    user         793K Jul 12 19:56 en.vcb.classes
-rw-rw-r--    1 koehn    user         614K Jul 12 19:56 en.vcb.classes.cats
-rw-rw-r--    1 koehn    user          94M Jul 12 18:08 euro.de
-rw-rw-r--    1 koehn    user          84M Jul 12 18:08 euro.en

giza.de-en:
total 422M
-rw-rw-r--    1 koehn    user         107M Jul 13 03:57 de-en.A3.final.gz
-rw-rw-r--    1 koehn    user         314M Jul 12 20:11 de-en.cooc
-rw-rw-r--    1 koehn    user         2.0K Jul 12 20:11 de-en.gizacfg

giza.en-de:
total 421M
-rw-rw-r--    1 koehn    user         107M Jul 13 11:03 en-de.A3.final.gz
-rw-rw-r--    1 koehn    user         313M Jul 13 04:07 en-de.cooc
-rw-rw-r--    1 koehn    user         2.0K Jul 13 04:07 en-de.gizacfg
```

```
model:
total 2.1G
-rw-rw-r--    1 koehn    user          94M Jul 13 19:59 aligned.de
-rw-rw-r--    1 koehn    user          84M Jul 13 19:59 aligned.en
-rw-rw-r--    1 koehn    user          90M Jul 13 19:59 aligned.grow-diag-final
-rw-rw-r--    1 koehn    user         214M Jul 13 20:33 extract.gz
-rw-rw-r--    1 koehn    user         212M Jul 13 20:35 extract.inv.gz
-rw-rw-r--    1 koehn    user          78M Jul 13 20:23 lex.f2n
-rw-rw-r--    1 koehn    user          78M Jul 13 20:23 lex.n2f
-rw-rw-r--    1 koehn    user          862 Jul 13 21:49 pharaoh.ini
-rw-rw-r--    1 koehn    user         1.2G Jul 13 21:49 phrase-table
```

Summary

- `-root-dir` – root directory, where output files are stored
- `-corpus` – corpus, expected in `$ROOT/corpus`
- `-e` – extension of the English corpus file
- `-f` – extension of the foreign corpus file
- `-lm` – language model file

6.3.2 Factored Translation Model Settings

More on factored translation models in the Overviev (page 75).

Summary

- `-alignment-factors` –
- `-translation-factors` –
- `-reordering-factors` –
- `-generation-factors` –
- `-decoding-steps` –

6.3.3 Lexicalized Reordering Model

More on lexicalized reording on the description of Training step 7: build reordering model (page 87).

Summary

- `-reordering` –
- `-reordering-smooth` –

6.3.4 Partial Training

You may have better ideas how to do word alignment, extract phrases or score phrases. Since the training is modular, you can start training at any of the seven training steps `-first-step` and end it at any subsequent step `-last-step`.

Again, the nine training steps are:

1. Prepare data
2. Run GIZA++
3. Align words
4. Get lexical translation table
5. Extract phrases
6. Score phrases
7. Build reordering model
8. Build generation models
9. Create configuration file

For instance, if you may have your own method to generate a word alignment, you want to skip these training steps and start with lexical translation table generation, you may specify this by

```
train-phrase-model.perl [...] --first-step 4
```

Summary

- `-first-step` – first step in the training process (default 1)
- `-last-step` – last step in the training process (default 7)

6.3.5 File Locations

A number of parameters allow you to break out of the rigid file name conventions of the training script. A typical use for this is that you want to try alternative training runs, but there is no need to repeat all the training steps.

For instance, you may want to try an alternative alignment heuristic. There is no need to rerun GIZA++. You could copy the necessary files from the `corpus` and the `giza.*` directories into a new root directory, but this takes up a lot of additional disk space and makes the file organization unnecessarily complicated.

Since you only need a new model directory, you can specify this with the parameter `-model-dir`, and stay within the precious root directory structure:

```
train-phrase-model.perl [...] --first-step 3 --alignment union --model-dir model-union
```

The other parameters for file and directory names fullfill similar purposes.

Summary

- `-corpus-dir` – corpus directory (default `$ROOT/corpus`)
- `-lexical-dir` – lexical translation probability directory (default `$ROOT/model`)
- `-model-dir` – model directory (default `$ROOT/model`)
- `-extract-file` – extraction file (default `$ROOT/model/extract`)
- `-giza-f2e` – GIZA++ directory (default `$ROOT/giza.`
  $F -$
  `E`$\}$
- `-giza-e2f` – inverse GIZA++ directory (default `$ROOT/giza.`
  $E -$
  `F`)

6.3.6 Alignment Heuristic

A number of different word alignment heuristics are implemented, and can be specified with the parameter `-alignment`. The options are:

- `intersect` – the intersection of the two GIZA++ alignments is taken. This usually creates a lot of extracted phrases, since the unaligned words create a lot of freedom to align phrases.
- `union` – the union of the two GIZA++ alignments is taken
- `grow-diag-final` – the default heuristic
- `grow-diag` – same as above, but without a call to function `FINAL()` (see background to word alignment).
- `grow` – same as above, but with a different definition of *neighboring*. Now diagonally adjacent alignment points are excluded.
- `grow` – no diagonal neighbors, but with `FINAL()`

Different heuristic may show better performance for a specific language pair or corpus, so some experimentation may be useful.

Summary

- `-alignment` – heuristic used for word alignment: intersect, union, grow, grow-final, grow-diag, grow-diag-final (default)

6.3.7 Maximum Phrase Length

The maximum length of phrases is limited to 7 words. The maximum phrase length impacts the size of the phrase translation table, so shorter limits may be desirable, if phrase table size is an issue. Previous experiments have shown that performance increases only slightly when including phrases of more that 3 words.

Summary

- `-max-phrase-length` – maximum length of phrases entered into phrase table (default 7)

6.3.8 GIZA++ Options

GIZA++ takes a lot of parameters to specify the behavior of the training process and limits on sentence length, etc. Please refer to the corresponding documentation for details on this.

Parameters can be passed on to GIZA++ with the switch `-giza-option`.

For instance, if you want to the change the number of iterations for the different IBM Models to 4 iterations of Model 1, 0 iterations of Model 2, 4 iterations of the HMM Model, 0 iterations of Model 3, and 3 iterations of Model 4, you can specify this by

```
train-phrase-model.perl [...] --giza-option m1=4,m2=0,mh=4,m3=0,m4=3
```

Summary

- `-giza-option` – additional options for GIZA++ training

6.3.9 Dealing with large training corpora

Training on large training corpora may become a problem for the GIZA++ word alignment tool. Since it stores the word translation table in memory, the size of this table may become too large for the available RAM of the machine. For instance, the data sets for the NIST Arabic-English and Chinese-English competitions require more than 4 GB of RAM, which is a problem for current 32-bit machines.

This problem can be remedied to some degree by a more efficient data structure in GIZA++, which requires the run of `snt2cooc` in advance on the corpus in parts and the merging on the

resulting output. All you need to know is that running the training script with the option `-parts n`, e.g. `-parts 3` may allow you to train on a corpus that was too large for a regular run.

Somewhat related to this problem caused by large training corpora is the problem of the large run time of GIZA++. It is possible to run the two GIZA++ separately on two machines with the switch `-direction`. When running one of the runs on one machine with `-direction 1` and the other run on a different machine or CPU with `-direction 2`, the processing time for training step 2 can be cut in half.

Summary

- `-parts` – prepare data for GIZA++ by running `snt2cooc` in parts
- `-direction` – run training step 2 only in direction 1 or 2 (for parallelization)