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).

November 23, 2012

Acknowledgments

The Moses decoder was supported by the European Framework 6 projects EuroMatrix, TC-Star, the European Framework 7 projects EuroMatrixPlus, Let's MT, META-NET and MosesCore 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

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.

1.2 Overview

1.2.1 Technology

Moses is an implementation of the **statistical** (or data-driven) approach to machine translation (MT). This is the dominant approach in the field at the moment, and is employed by the online translation systems deployed by the likes of Google and Microsoft. In statistical machine translation (SMT), translation systems are **trained** on large quantities of **parallel** data (from which the systems learn how to translate small segments), as well as even larger quantities of monolingual data (from which the systems learn what the target language should look like). Parallel data is a collection of sentences in two different languages, which is **sentence-aligned**, in that each sentence in one language is matched with its corresponding translated sentence in the other language. It is also known as a **bitext**.

The training process in Moses takes in the parallel data and uses coocurrences of words and segments (known as **phrases**) to infer translation correspondences between the two languages of interest. In **phrase-based machine translation**, these correspondences are simply between continuous sequences of words, whereas in **hierarchical phrase-based machine translation** or **syntax-based translation**, more structure is added to the correspondences. For instance a hierarchical MT system could learn that the German *hat X gegessen* corresponds to the English *ate X*, where the *X*s are replaced by any German-English word pair. The extra structure used in these types of systems may or may not be derived from a linguistic analysis of the parallel data. Moses also implements an extension of phrase-based machine translation know as **factored translation** which enables extra linguistic information to be added to a phrase-based systems.

For more information about the Moses translation models, please refer to the tutorials on phrase-based MT (page **??**), syntactic MT (page **??**) or factored MT (page **??**).

Whichever type of machine translation model you use, the key to creating a good system is lots of good quality data. There are many free sources of parallel data[1] which you can use to train sample systems, but (in general) the closer the data you use is to the type of data you want to translate, the better the results will be. This is one of the advantages to using on open-source tool like Moses, if you have your own data then you can tailor the system to your needs and potentially get better performance than a general-purpose translation system. Moses needs sentence-aligned data for its training process, but if data is aligned at the document level, it can often be converted to sentence-aligned data using a tool like hunalign[2]

1.2.2 Components

The two main components in Moses are the **training pipeline** and the **decoder**. There are also a variety of contributed tools and utilities. The training pipeline is really a collection of tools (mainly written in perl, with some in C++) which take the raw data (parallel and monolingual) and turn it into a machine translation model. The decoder is a single C++ application which, given a trained machine translation model and a source sentence, will translate the source sentence into the target language.

The Training Pipeline

There are various stages involved in producing a translation system from training data, which are described in more detail in the training documentation (page **??**) and in the baseline system guide (page **??**). These are implemented as a pipeline, which can be controlled by the Moses experiment management system (page **??**), and Moses in general makes it easy to insert different types of external tools into the training pipeline.

The data typically needs to be prepared before it is used in training, tokenising the text and converting tokens to a standard case. Heuristics are used to remove sentence pairs which look to be misaligned, and long sentences are removed. The parallel sentences are then **word-aligned**, typically using GIZA++[3], which implements a set of statistical models developed at IBM in the 80s. These word alignments are used to extract phrase-phrase translations, or hierarchical rules as required, and corpus-wide statistics on these rules are used to estimate probabilities.

An important part of the translation system is the **language model**, a statistical model built using monolingual data in the target language and used by the decoder to try to ensure the fluency of the output. Moses relies on external tools for language model building, described here (page **??**).

The final step in the creation of the machine translation system is **tuning** (page **??**), where the different statistical models are weighted against each other to produce the best possible translations. Moses contains implementations of the most popular tuning algorithms.

The Decoder

[1] http://www.statmt.org/moses/?n=Moses.LinksToCorpora

[2] http://mokk.bme.hu/resources/hunalign/

[3] http://code.google.com/p/giza-pp/

The job of the Moses decoder is to find the highest scoring sentence in the target language (according to the translation model) corresponding to a given source sentence. It is also possible for the decoder to output a ranked list of the translation candidates, and also to supply various types of information about how it came to its decision (for instance the phrase-phrase correspondences that it used).

The decoder is written in a modular fashion and allows the user to vary the decoding process in various ways, such as:

- **Input:** This can be a plain sentence, or it can be annotated with xml-like elements to guide the translation process, or it can be a more complex structure like a lattice or confusion network (say, from the output of speech recognition)
- **Translation model:** This can use phrase-phrase rules, or hierarchical (perhaps syntactic rules). It can be compiled into a binarised form for faster loading. It can be supplemented with **features** to add extra information to the translation process, for instance features which indicate the sources of the phrase pairs in order to weight their reliability.
- **Decoding algorithm:** Decoding is a huge search problem, generally too big for exact search, and Moses implements several different strategies for this search, such as stack-based, cube-pruning, chart parsing etc.
- **Language model:** Moses supports several different language model toolkits (SRILM, KenLM, IRSTLM, RandLM) each of which has there own strengths and weaknesses, and adding a new LM toolkit is straightforward.

The Moses decoder also supports multi-threaded decoding (since translation is *embarassingly parallelisable*[4]), and also has scripts to enable multi-process decoding if you have access to a cluster.

Contributed Tools

There are many contributed tools in Moses which supply additional functionality over and above the standard training and decoding pipelines. These include:

- **Moses server:** which provides an xml-rpc interface to the decoder
- **Web translation:** A set of scripts to enable Moses to be used to translate web pages
- **Analysis tools:** Scripts to enable the analysis and visualisation of Moses output, in comparison with a reference.

There are also tools to evaluate translations, alternative phrase scoring methods, an implementation of a technique for weighting phrase tables, a tool to reduce the size of the phrase table, and other contributed tools.

1.2.3 Development

Moses is an open-source project, licensed under the LGPL[5], which incorporates contributions from many sources. There is no formal management structure in Moses, so if you want to contribute then just mail support[6] and take it from there. There is a list[7] of possible projects on this website, but any new MT techiques are fair game for inclusion into Moses.

[4] http://en.wikipedia.org/wiki/Embarrassingly_parallel

[5] http://www.gnu.org/copyleft/lesser.html

[6] http://www.statmt.org/moses/?n=Moses.MailingLists

[7] http://www.statmt.org/moses/?n=Moses.GetInvolved

In general, the Moses administrators are fairly open about giving out push access to the git repository, preferring the approach of removing/fixing bad commits, rather than vetting commits as they come in. This means that trunk occasionally breaks, but given the active Moses user community, it doesn't stay broken for long. The nightly builds and tests reported on the cruise control[8] web page will be ramped up in code and platform coverage through 2012 in order to reduce the likelihood of breakages going unnoticed.

1.2.4 Moses in Use

The liberal licensing policy in Moses, together with its wide coverage of current SMT technology and complete tool chain, make it probably the most widely used open-source SMT system. It is used in teaching, research, and, increasingly, in commercial settings.

Commercial use of Moses is promoted and tracked by TAUS[9]. The most common current use for SMT in commercial settings is **post-editing** where machine translation is used as a first-pass, with the results then being edited by human translators. This can often reduce the time (and hence total cost) of translation. There is also work on using SMT in **computer-aided translation**, but at the moment (April 2012) this is more of a research topic, for example in the two recently launched EU projects, Casmacat[10] and MateCat[11].

1.2.5 History

2005	Hieu Hoang (then student of Philipp Koehn) starts Moses as successor to Pharoah
2006	Moses is the subject of the JHU workshop, first check-in to public repository
2006	Start of Euromatrix, EU project which helps fund Moses development
2007	First machine translation marathon held in Edinburgh
2009	Moses receives support from EuromatrixPlus, also EU-funded
2010	Moses now supports hierarchical and syntax-based models, using chart decoding
2011	Moses moves from sourceforge to github, after over 4000 sourceforge check-ins
2012	EU-funded MosesCore launched to support continued development of Moses

Subsection last modified on April 24, 2012, at 04:19 PM

1.3 Work-in-Progress

Things that are happening in Moses since the last release. Lots of todo's so if you fancy helping out, email someone.

[8] http://www.statmt.org/moses/cruise/

[9] http://www.translationautomation.com/user-cases/machines-takes-center-stage.html

[10] http://www.casmacat.eu/

[11] http://www.matecat.com/

<th align='center'>Feature</th><th align='center'>Finished coding</th><th align='center'>Tested</th><th a
Link to tc malloc for fast C++ execution
Include Word alignment on by default during training and decoding
Integrating Phi's TM-MT into Moses
Integrating Marcin's compressed pt into EMS. Regression test added
Testing cygwin build

Simplify feature function framework. Merge all [weights-*] sections in ini file

Lattice decoding in chart decoding

Sparse feature

1.4 Releases

1.4.1 Release 0.91 (12th October, 2012)

Get the code on **github**[12]
Pre-made **models**[13] trained using this version has also been released

Tested on 8 Europarl language pairs, phrase-based, hierarchical, and phrase-base factored models. All runs through without major intervention. Known issues:

1. Hierarchical models crashes on evaluation when threaded. Strangely, run OK during tuning
2. EMS bugs when specifying multiple language models
3. Complex factored models not tested
4. Hierarchical models with factors does not work

11th July, 2012

(Not a code release, just a roundup of the new features that has been implemented in the past year:

[12] https://github.com/moses-smt/mosesdecoder/tree/RELEASE-0.91
[13] http://www.statmt.org/moses/RELEASE-0.91/

<th align='center'>Feature</th><th align='center'>Finished coding</th><th align='center'>Tested</th><

Lexi Birch's LR score integrated into tuning

Asynchronous, batched LM requests for phrase-based models

multithreaded tokenizer

KB Mira

Training & decoding more resilient to non-printing characters and Moses' reserved characters. Escaping the

Simpler installation

Factors works with chart decoding

Less IO and disk space needed during training. Everything written directly to gz files

Parallel training

Adam Lopez's suffix array integrated into Moses's training & decoding

Major MERT code cleanup

Wrapper for berkeley parser (german)

Option to use p(RHS_t|RHS_s,LHS) or p(LHS,RHS_t|RHS_s), as a grammar rule's direct translation score

optional PCFG scoring feature for target syntax models

add -snt2cooc option to use mgiza's reduced memory snt2cooc program

queryOnDiskPt program

Output phrase segmentation to n-best when -report-segmentation is used.

CDER and WER metric in tuning

Lossy Distributed Hash Table Language Model

InterpolatedScorer for mert

IRST LM training integrated into Moses

GlobalLexiconModel

tmcombine (translation model combination)

Alternative to CKY+ for scope-3 grammar. Reimplementation of Hopkins and Langmead (2010)

Sample java client for moses server

support for mgiza, without having to install giza++ as well.

interpolated language models

Duplicate removal in MERT

Use bjam instead of automake to compile

Recaser train script updated to support IRSTLM as well

extract-ghkm

PRO tuning algorithm

Cruise control

Faster SCFG rule table format

LM OOV feature

Ter Scorer in mert

multi-threading for decoder & mert

expose n-gram length as part of LM state calculation

Changes to chart decoder cube pruning: create one cube per dotted rule instead of one per translation

Syntactic LM

Czech detokenization

13th August, 2010

Changes since 9th August, 2010

1. change or delete character Ø to 0 in extract-rules.cpp (Raphael and Hieu)

9th August, 2010

Changes since 24th April, 2010:

1. Add option of retaining alignment information in the phrase-based phrase table. Decoder loads this information if present. (Hieu Hoang & Raphael Payen)
2. When extracting rules, if the source or target syntax contains an unsupported escape sequence (anything other than "<", ">", "&", "&apos", and "&quot") then write a warning message and skip the sentence pair (instead of asserting).
3. In bootstrap-hypothesis-difference-significance.pl, calculates the p-value and confidence intervals not only using BLEU, but also the NIST score. (Mark Fishel)
4. Dynamic Suffix Arrays (Abbey levenberg)
5. Merge multithreaded moses into moses (Barry Haddow)
6. Continue partial translation (Ondrej Bojar and Ondrej Odchazel)
7. Bug fixes, minor bits & bobs. (Philipp Koehn, Christian Hardmeier, Hieu Hoang, Barry Haddow, Phil Williams, Ondrej Bojar, Abbey, Mark Mishel, Lane Schwartz, Nicola Bertoldi, Raphael, ...)

26th April, 2010

Changes since the last time:

1. Synchronous CFG based decoding, a la Hiero (Chiang 2005), plus with syntax. And all the scripts to go with it. (Thanks to Philip Williams and Hieu Hoang)
2. caching clearing in IRST LM (Nicola Bertoldi)
3. Factored Language Model. (Ondrej Bojar)
4. Fixes to lattice (Christian Hardmeier, Arianna Bisazza, Suzy Howlett)
5. zmert (Ondrej Bojar)
6. Suffix arrays (Abby Levenberg)
7. Lattice MBR and consensus decoding (Barry Haddow & Abhishek Arun)
8. Simple program that illustrates how to access a phrase table on disk from an external program (Felipe Sánchez-Martínez)
9. Odds and sods by Raphael Payen and Sara Stymne.

1st April, 2010

Hi All

Seems like the last release uncovered some bugs hidden in the woodwork. So for you delectation, we are rolling out a special April Fool's service pack.

Fixes:

1. Fix for Visual Studio, and potentially other compilers (thanks to Barry, Christian, Hieu)
2. Mem leak in unique n-best (thanks to Barry)
3. Makefile fix for moses server (thanks to Barry)

Available now as a download, svn up, and from your local high-street stockist.

26th March, 2010

Changes since the last time:

1. minor bug fixes & tweaks, especially to the decoder, MERT scripts (thanks to too many people to mention)

2. fixes to make decoder compile with most versions of gcc, Visual studio and other compilers (thanks to Tom Hoar, Jean-Bapist Fouet).
3. multi-threaded decoder (thanks to Barry Haddow)
4. update for IRSTLM (thanks to nicola bertoldi & Marcello Federico)
5. run mert on a subset of features (thanks to nicola bertoldi)
6. Training using different alignment models (thanks to Mark Fishel)
7. "a handy script to get many translations from Google" (thanks to Ondrej Bojar)
8. Lattice MBR (thanks to Abhishek Arun & Barry Haddow)
9. Option to compile moses as a dynamic library (thanks to Jean-Bapist Fouet).
10. hierarchical re-ordering model (thanks to Christian Harmeier, Sara Styme, Nadi, Marcello, Ankit Srivastava, Gabriele Antonio Musillo, Philip Williams, Barry Haddow).
11. Global Lexical re-ordering model (thanks to Philipp Koehn)
12. Experiment.perl scripts for automating the whole MT pipeline (thanks to Philipp Koehn)

.... (There's earlier releases, they're still on sourceforge if you want to take a trip down memory lane)

```
http://sourceforge.net/projects/mosesdecoder/files/mosesdecoder/
```

Subsection last modified on November 23, 2012, at 10:20 AM

1.5 Getting Started with Moses

This section will show you how to install and build Moses, and how to use Moses to translate with some simple models. If you experience problems, then please check the common pitfalls at the bottom of this page, and if this doesn't solve the problem then mail support[14].

1.5.1 Platforms

The primary development platform for Moses is Linux, and this is the recommended platform since you will find it easier to get support for it. However Moses does work on other platforms:

1.5.2 Windows Installation

Moses can run on Windows under Cygwin. Installation is exactly the same as for Linux and Mac. (Are you running it on Windows? If so, please give us feedback on how it works).

1.5.3 OSX Installation

This is also possible and widely used by Moses developers. The old step-by-step guide[15] has some hints about OSX installation.

[14] http://www.statmt.org/moses/?n=Moses.MailingLists

[15] http://www.statmt.org/moses_steps.html

1.5.4 Linux Installation

Install boost

Moses requires boost[16]. Your distribution probably has a package for it. If your distribution has separate development packages, you need to install those too. For example, Ubuntu requires `libboost-all-dev`.

Check out the source code

The source code is stored in a git repository on github[17].

You can clone this repository with the following command (the instructions that follow from here assume that you run this command from your home directory):

```
git clone git://github.com/moses-smt/mosesdecoder.git
```

1.5.5 Compile

After you check out from git, examine the options you want.

```
cd ~/mosesdecoder
./bjam --help
```

For example, if you have 8 CPUs, build in parallel:

```
./bjam -j8
```

See ~/mosesdecoder/BUILD-INSTRUCTIONS.txt for more information.

1.5.6 bjam options

This is a list of options to bjam. On a system with Boost installed in a standard path, none should be required, but you may want additional functionality or control.

Optional packages

Language models

[16] http://www.boost.org/

[17] https://github.com/moses-smt/mosesdecoder

In addition to KenLM and ORLM (which are always compiled):

<dl><dt>–with-irstlm=/path/to/irstlm</dt><dd>Path to IRSTLM installation </dd><dt>–with-randlm=/path/to/randlm</dt><dd>Path to RandLM installation </dd><dt>–with-srilm=/path/to/sr to SRILM installation.

If your SRILM install is non-standard, use these options:

<dl><dt>–with-srilm-dynamic</dt><dd>Link against srilm.so. </dd><dt>–with-srilm-arch=arch</dt><dd the arch setting given by /path/to/srilm/sbin/machine-type

Other packages

<dl><dt>–with-boost=/path/to/boost</dt><dd>If Boost is in a non-standard location, specify it here. This directory is expected to contain include and lib or lib64. </dd><dt>–with-xmlrpc-c=/path/to/xmlrpc-c</dt><dd>Specify a non-standard libxmlrpc-c installation path. Used by Moses server. </dd><dt>–with-cmph=/path/to/cmph</dt><dd>Path where CMPH is installed. Used by the compact phrase table and compact lexical reordering table. </dd><dt>–with-tcmalloc</dt><dd>Use thread-caching malloc. </dd><dt>–with-regtest=/path/to/moses-regression-tests</dt><dd>Run the regression tests using data from this directory. Tests can be downloaded from https://github.com/moses-smt/moses-regression-tests.

Installation

<dl><dt>–prefix=/path/to/prefix</dt><dd>sets the install prefix [default is source root]. </dd><dt>–bindir=/path/to/prefix/bin</dt><dd>sets the bin directory [PREFIX/bin] </dd><dt>–libdir=/path/to/pre the lib directory [PREFIX/lib] </dd><dt>–includedir=/path/to/prefix/include</dt><dd>installs headers. Does not install if missing. No argument defaults to PREFIX/include . </dd><dt>–install-scripts=/path/to/scripts</dt><dd>copies scripts into a directory. Does not install if missing. No argument defaults to PREFIX/scripts . </dd><dt>–git</dt><dd> appends the git revision to the prefix directory.

Build Options

By default, the build is multi-threaded, optimized, and statically linked.

<dl><dt>threading=single|multi</dt><dd>controls threading (default multi) </dd><dt>variant=release|deb optimized (default), for debug, or for profiling </dd><dt>link=static|shared</dt><dd>controls preferred linking (default static) </dd><dt>–static</dt><dd>forces static linking (the default will fall back to shared) </dd><dt>debug-symbols=on|off</dt><dd>include (default) or exclude debugging information also known as -g </dd><dt>–notrace</dt><dd>compiles without TRACE macros </dd><dt>–enable-boost-pool</dt><dd>uses Boost pools for the memory SCFG table </dd><dt>–enable-mpi</dt><dd>switch on mpi </dd><dt>–without-libsegfault</dt><dd>doe not link with libSegFault </dd><dt>–max-kenlm-order</dt><dd>maximum ngram order that kenlm can process (default 6) </dd><dt>–max-factors</dt><dd>maximum number of factors (default 4)

Controlling the Build

<dl><dt>-q</dt><dd>quit on the first error </dd><dt>-a</dt><dd>to build from scratch </dd><dt>-j$NCPUS</dt><dd>to compile in parallel </dd><dt>–clean</dt><dd>to clean

1.5.7 Run it for the first time

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

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

Note that the configuration file `moses.ini` in each directory is set to use KenLM language model toolkit by default. If you prefer to use another LM toolkit, edit the language model entry in `moses.ini` and replace the first number as follows:

for SRI (requires separate installation and `./bjam -with-srilm`)

```
[lmodel-file]
0 0 3 lm/europarl.srilm.gz
```

For IRST (requires separate installation and `./bjam -with-irstlm`)

```
[lmodel-file]
1 0 3 lm/europarl.srilm.gz
```

For KenLM (compiled by default)

```
[lmodel-file]
8 0 3 lm/europarl.srilm.gz
```

Look here[18] for more details.

Run the deocder

```
cd ~/mosesdecoder/sample-models
~/mosesdecoder/bin/moses -f phrase-model/moses.ini < phrase-model/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`).

[18] http://www.statmt.org/moses/?n=FactoredTraining.BuildingLanguageModel#ntoc1

1.5.8 Chart Decoder

The chart decoder is created as a separate executable:

```
~/mosesdecoder/bin/moses_chart
```

You can run the chart demos from the sample-models directory as follows

```
~/mosesdecoder/bin/moses_chart -f string-to-tree/moses.ini < string-to-tree/in > out.stt
~/mosesdecoder/bin/moses_chart -f tree-to-tree/moses.ini < tree-to-tree/in.xml > out.ttt
```

The expected result of the string-to-tree demo is

```
this is a small house
```

1.5.9 Next Steps

Why not try to build a Baseline (page **??**) translation system with freely available data?

1.5.10 Common Pitfalls

Subsection last modified on November 04, 2012, at 07:05 AM

1.6 Get Involved

Moses is an open source project that is at home in the academic research community. There are several venues where this community gathers, such as:

- The main conferences in the field: ACL, EMNLP, MT Summit, etc.
- The annual ACL Workshop on Statistical Machine Translation[19]
- The annual Machine Translation Marathon[20]

[19] http://www.statmt.org/wmt12/

[20] http://www.statmt.org/moses/?n=Moses.Marathons

The main forum for communication on Moses is the Moses support mailing list[21].

Moses is being developed as a reference implementation of state-of-the-art methods in statistical machine translation. Extending this implementation may be the subject of undergraduate or graduate theses, or class projects. Typically, developers extend functionality that they required for their projects, or to explore novel methods. Let us know, if you made an improvement, no matter how minor. Let us also know if you found or fixed a bug.

We are aware of some commercial deployments of Moses, for instance as described by TAUS[22]. Please let us know if you use Moses commercially. Do not hesitate to contact the core developers of Moses. They are willing to answer questions and may be even available for consulting services.

If you are looking for projects to improve Moses, please consider the following list:

1.6.1 Chart Decoding

- **Decoding algorithms for syntax-based models:** Moses generally supports a large set of grammar types. For some of these (for instance ones with source syntax, or a very large set of non-terminals), the implemented CKY decoding algorithm is not optimal. Implementing search algorithms for dedicated models, or just to explore alternatives, would be of great interest.
- **Cube pruning for factored models:** Complex factored models with multiple translation and generation steps push the limits of the current factored model implementation which exhaustively computes all translations options up front. Using ideas from cube pruning (sorting the most likely rules and partial translation options) may be the basis for more efficient factored model decoding.
- **Missing features for chart decoder:** a number of features are missing for the chart decoder, namely: MBR decoding (should be simple), lattice decoding, and output of word alignments. In general, reporting and analysis within experiment.perl could be improved.
- **More efficient rule table for chart decoder:** The in-memory rule table for the hierarchical decoder loads very slowly and uses a lot of RAM. A optimized implementation that is vastly more efficient on both fronts should be feasible.

1.6.2 Phrase-based Models

- **Faster training for the global lexicon model:** Moses implements the global lexicon model proposed by Mauser et al. (2009), but training features for each target word using a maximum entropy trainer is very slow (years of CPU time). More efficient training or accommodation of training of only frequent words would be useful.
- **A better phrase table:** The current binarised phrase table suffers from (i) far too many layers of indirection in the code making it hard to follow and inefficient (ii) a cache-locking mechanism which creates excessive contention; and (iii) lack of extensibility meaning that (e.g.) word alignments were added on by extensively duplicating code. A new phrase table could make Moses faster and more extensible.

[21] http://www.statmt.org/moses/?n=Moses.MailingLists

[22] http://www.translationautomation.com/user-cases/machines-takes-center-stage.html

1.6.3 Sparse Moses

(see SparseFeatureFunctions (page **??**))

- **Optimisation of sparse Moses:** When you add a lot of features it significantly slows down the decoder and increases memory usage. If sparse Moses could be made more efficient then it could be merged back into the trunk. One idea is to avoid storing feature vectors wherever possible, but to just store the total score of hypotheses, only calculating full feature vectors when they are required for, e.g., n-best lists.

1.6.4 Hybrid Translation

- **Integration with translation memories:** We implemented an offline version of converting translation memory matches into very large hierarchical rules (Koehn and Senellart, 2011). It should not be too hard to integrate this into the decoder on top of the suffix array phrase table implementation.

1.6.5 Engineering Improvements

- **Parellelisation of Training:** MT Training is slow process, so parallelising as much of it as possible is a route to speeding it up. See this call[23]
- **Better Disk Usage in Training** Moses training uses a lot of disk during training, probably unnecessarily, so it would be good to see if this could be minimised. See this thread[24]
- **Integration of sigfilter:** The filtering algorithm of Johnson et al is available[25] in Moses, but it is not well integrated, has awkward external dependencies and so is seldom used. At the moment the code is in the contrib directory. A useful project would be to refactor this code to use the Moses libraries for suffix arrays, and to integrate it with the Moses experiment management system (ems). The goal would be to enable the filtering to be turned on with a simple switch in the ems config file.
- **Boostification:** Moses has allowed boost[26] since Autumn 2011, but there are still many areas of the code that could be improved by usage of the boost libraries, for instance using shared pointers in collections.
- **Unit-testing:** The core of Moses is completely lacking in unit testing, although some exist for MERT and KenLM (using boost test). Increasing test coverage is a priority for 2012, and implementing unit tests is a good way of learning about the code ;-). Some refactoring will be necessary in order to make Moses "unit-testable".
- **Cruise control** Moses has cruise control[27] running on a server at the University of Edinburgh, however this only tests one platform (OpenSuse). If you have a different platform, and care about keeping Moses stable on that platform, then you could set up a cruise control instance too. The code is all in the standard Moses distribution.
- **Improved feature-function interface** Adding a new feature function is now much easier because of efforts (page **??**) at the 2009 machine translation marathon. However it's still harder than it should be, for instance dealing with features that should be cached in the

[23] http://comments.gmane.org/gmane.comp.nlp.moses.user/6464

[24] http://comments.gmane.org/gmane.comp.nlp.moses.user/6123

[25] http://www.statmt.org/moses/?n=Moses.AdvancedFeatures#ntoc16

[26] http://www.boost.org

[27] http://www.statmt.org/moses/cruise/

translation options, getting mert to recognise your feature, figuring out what all the different names mean. Also, the stateful interface demands a `Hypothesis` when it should just need a `TranslationOption` (this is because of legacy LM code).

1.6.6 Documentation

- **Maintenance** The documentation always needs maintenance as new features are introduced and old ones are updated. Such a large body of documentation inevitably contains mistakes and inconsistencies, so any help in fixing these would be most welcome. If you want to work on the documentation, just introduce yourself on the mailing list.
- **Help Messages** Moses has a lot of executables, and often the help messages are quite cryptic or missing. A help message in the code is more likely to be maintained than separate documentation, and easier to locate when you're trying to find the right options. Fixing the help messages would be a useful contribution to making Moses easier to use.

1.6.7 Ongoing projects (may need some outside help)

- **Online discriminative training:** *(Eva Hasler)* MERT optimization does not support more than a few features. There has been recent progress on using online learning algorithms (e.g., MIRA) to support thousands of features. Implementing such an algorithm in Moses would be very useful.
- **Analysis of results:** *(Philipp Koehn)* Assessing the impact of variations in the design of a machine translation system by observing the fluctuations of the BLEU score may not be sufficiently enlightening. Having more analysis of the types of errors a system makes should be very useful.
- **Incremental updating of translation and language model:** *(Abby Levenberg)* This work is basically completed and will be properly documented shortly.

1.6.8 Recently completed projects

- **Extending multi-threading to RandLM, IRSTLM, and tree-based decoding:** Multi-threading is very useful if you have a multi-core computer. While Moses generally supports multi-threading, some implemented features of Moses, however, are not yet thread-safe.

Subsection last modified on June 04, 2012, at 01:07 PM

2

Tutorials

2.1 Phrase-based Tutorial

This tutorial describes the workings of the phrase-based decoder in Moses, using a simple model downloadable from the Moses website.

2.1.1 Content

- A Simple Translation Model (page **??**)
- Running the Decoder (page **??**)
- Trace (page **??**)
- Verbose (page **??**)
- Tuning for Quality (page **??**)
- Tuning for Speed (page **??**)
- Translation Table Size (page **??**)
- Hypothesis Stack Size (Beam) (page **??**)
- Limit on Distortion (Reordering) (page **??**)

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, and
- `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 (it needs to be run from the `sample-models` directory) :

```
% echo ’das ist ein kleines haus’ | moses -f phrase-model/moses.ini > out
Defined parameters (per moses.ini or switch):
config: phrase-model/moses.ini
input-factors: 0
lmodel-file: 8 0 3 lm/europarl.srilm.gz
mapping: T 0
n-best-list: nbest.txt 100
ttable-file: 0 0 0 1 phrase-model/phrase-table
ttable-limit: 10
weight-d: 1
weight-l: 1
weight-t: 1
weight-w: 0
Loading lexical distortion models...have 0 models
Start loading LanguageModel lm/europarl.srilm.gz : [0.000] seconds
Loading the LM will be faster if you build a binary file.
Reading lm/europarl.srilm.gz
----5---10---15---20---25---30---35---40---45---50---55---60---65---70---75---80---85---90---95--100
****************************************************************************************************
The ARPA file is missing <unk>.  Substituting log10 probability -100.000.
Finished loading LanguageModels : [2.000] seconds
```

```
Start loading PhraseTable phrase-model/phrase-table : [2.000] seconds
filePath: phrase-model/phrase-table
Finished loading phrase tables : [2.000] seconds
Start loading phrase table from phrase-model/phrase-table : [2.000] seconds
Reading phrase-model/phrase-table
----5---10---15---20---25---30---35---40---45---50---55---60---65---70---75---80---85---90---95--100
****************************************************************************************************
Finished loading phrase tables : [2.000] seconds
IO from STDOUT/STDIN
Created input-output object : [2.000] seconds
Translating line 0  in thread id 0
Translating: das ist ein kleines haus

Collecting options took 0.000 seconds
Search took 0.000 seconds
BEST TRANSLATION: this is a small house [11111]  [total=-28.923] <<0.000, -5.000, 0.000, -27.091, -1.833>>
Translation took 0.000 seconds
Finished translating

% 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 0 1 phrase-model/phrase-table

# language models: type(srilm/irstlm), factors, order, file
[lmodel-file]
8 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 phrase-model/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 phrase-model/moses.ini < phrase-model/in > out
```

2.1.4 Trace

How the decoder works is described in detail in the background (page **??**) 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 phrase-model/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 word 2-2, `ein`,
- `small` was generated from the German word 3-3, `kleines`, and
- `house` was generated from the German word 4-4, `haus`.

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

```
echo 'ein haus ist das' | moses -f phrase-model/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 phrase-model/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, -27.091, -1.833
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), its 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 phrase-model/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-probability cost of -2.951 is accumulated. Together with the future cost estimate for the remaining part of 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 **??**) 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** ensures 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 models 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 is 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 phrase-model/moses.ini -d 0
this is a house
```

With the default weights, the translation comes out wrong:

```
% echo 'ein haus ist das' | moses -f phrase-model/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 phrase-model/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 phrase-model/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 **??**) 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 phrase-model/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 phrase-model/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 phrase-model/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 phrase-model/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 phrase-model/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 phrase-model/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 phrase-model/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.

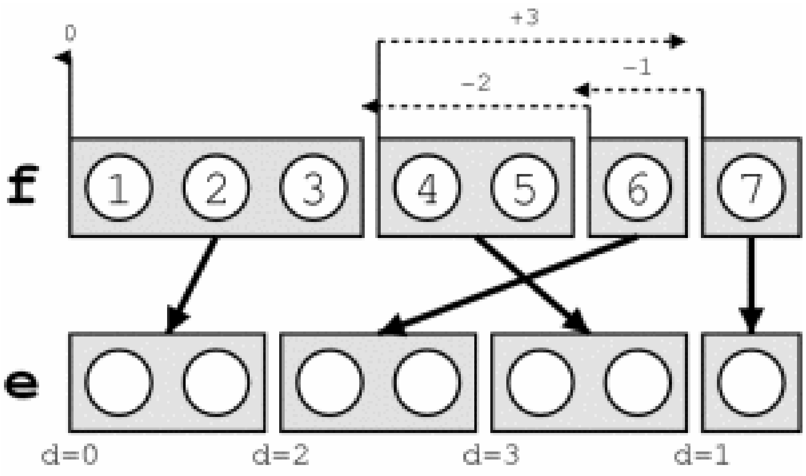
In this toy example, a worse translation is only generated with a stack size of 1. Again, by worse translation, we mean worse scoring according to our model (-30.991 vs. -28.923). If it is actually a worse translation in terms of translation quality, is another question. However, the task of the decoder 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 $D(e,f) = - \Sigma_i$ (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.

Subsection last modified on April 25, 2012, at 01:40 PM

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 an unfactored model (page **??**)
- Train a model with POS tags (page **??**)
- Train a model with generation and translation steps (page **??**)
- Train a morphological analysis and generation model (page **??**)
- Train a model with multiple decoding paths (page **??**)

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 an 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-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 `unfactored/model` in about 1 hour (?). For a quicker training run that only takes a few minutes (with much worse results) use the just the first 1000 sentence pairs of the corpus, contained in `factored-corpus/proj-syndicate.1000`.

```
% train-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, and
- part of speech with additional morphological information.

The English factors are

- surface form,
- lemma, and
- 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-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 0.0170876 2.718
[...]
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-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-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), and
- 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|vvfin putin|putin|nn|nn.masc.cas.sg .|.|per|per' > in
% moses -f morphgen/model/moses.ini < in
BEST TRANSLATION: a|a|dt individual|individual|nn describes|describe|vbz putin|putin|nnp .|.|. [11111]  [total=-17.269]
<<0.000, -5.000, 0.000, -38.631, -13.357, -2.773, -21.024, 0.000, -1.386, -1.796, -4.341, -3.189, -4.630, 4.999,
-13.478, -14.079, -4.911, -5.774, 4.999>>
```

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-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-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
```

Subsection last modified on April 19, 2012, at 04:27 PM

2.3 Syntax Tutorial

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.*

Exodus 15, 24-25

2.3.1 Content

- Tree-Based Models (page **??**)
- A Word on Terminology (page **??**)
- Chart Decoding (page **??**)
- Decoding (page **??**)
- Trace (page **??**)
- Rule Table (page **??**)
- More Example (page **??**)
- Decoder Parameters (page **??**)
- Beam Settings (page **??**)
- Limiting Reordering (page **??**)
- Handling Unknown Words (page **??**)
- Technical Settings (page **??**)
- Training (page **??**)

- Training Parameters (page **??**)
- Training Syntax Models (page **??**)
- Annotation Wrappers (page **??**)
- Relaxing Parses (page **??**)
- On-Disk Rule Table (page **??**)
- Common Pitfalls (page **??**)
- Using Meta-symbols in Non-terminal Symbols (e.g., CCG) (page **??**)
- Different Kinds of Syntax Models (page **??**)
- String-to-Tree (page **??**)
- Tree-to-string (page **??**)
- Input tree representation (page **??**)
- Translation rules (page **??**)
- Consequences (page **??**)
- Format of text rule table (page **??**)

Moses supports models that have become known as *hierarchical phrase-based models* and *syntax-based models*. These models use a grammar consisting of SCFG (Synchronous Context-Free Grammar) rules. In the following, we refer to these models as **tree-based models**.

2.3.2 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:

```
ne X1 pas -> not X1 (French-English)
ate X1 -> habe X1 gegessen (English-German)
X1 of the X2 -> le X2 X1 (English-French)
```

The variables in these grammar rules are called non-terminals, since their 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 a data structure is created that is a called a tree. To fully make this point, consider the following input and translation rules:

```
Input: Das Tor geht schnell auf
Rules: Das Tor -> The door
schnell -> quickly
geht X1 auf -> opens X1
X1 X2 -> X1 X2
```

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

```
X4                           X4
_|_                          _|_
/     \                      /     \
/        X3                  /        X3
/        / | \               /         / |
X1       /   X2  \           X1        /   X2
/  \    /     |     \        /  \     /     |
Das Tor geht schnell auf     The door opens quickly
```

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 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, and
- 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.

2.3.3 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`:

[1] http://www.statmt.org/moses/?n=Development.GetStarted#chart

[2] http://www.statmt.org/moses/download/sample-models.tgz

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

The decoder is called just as for phrase models:

```
% echo 'das ist ein haus' | moses_chart -f 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 11
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>>
```

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 hypothesis (`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`), and
- 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 `string-to-tree` 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 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` is an extension of the Pharaoh/Moses phrase-table, so it will be familiar to anybody who has used it before. Here are some lines as example:

```
gibt [X] ||| gives [ADJ] ||| 1.0 ||| ||| 3 5
es gibt [X] ||| there is [ADJ] ||| 1.0 |||  ||| 2 3
[X][DT] [X][NN] [X] ||| [X][DT] [X][NN] [NP] ||| 1.0 ||| 0-0 1-1 ||| 2 4
[X][DT] [X][ADJ] [X][NN] [X] ||| [X][DT] [X][ADJ] [X][NN] [NP] ||| 1.0 ||| 0-0 1-1 2-2 ||| 5 6
[X][V] [X][NP] [X] ||| [X][V] [X][NP] [VP] ||| 1.0 ||| 0-0 1-1 ||| 4 3
```

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

- the source string and source left-hand-side,
- the target string and target left-hand-side,
- the alignment between non-terminals (using word positions),
- score(s): here only one, but typically multiple scores are used, and
- frequency counts of source & target phrase (for debuggin purposes, not used during decoding).

The format is slightly different from the Hiero format. For example, the Hiero rule

```
[X] ||| [X,1] trace ’ ||| [X,1] &#52628;&#51201; ’ ||| 0.727273 0.444625 1 0.172348 2.718
```

is formatted as

```
[X][X] trace ’ [X] ||| [X][X] &#52628;&#51201; ’ [X] ||| 0.727273 0.444625 1 0.172348 2.718 ||| 0-0 ||| 2 3
```

A syntax rule in a string-to-tree grammar:

```
[NP] ||| all [NN,1] ||| &#47784;&#46304; [NN,1] ||| 0.869565 0.627907 0.645161 0.243243 2.718
```

is formatted as

```
all [X][NN] [X] ||| &#47784;&#46304; [X][NN] [NP] ||| 0.869565 0.627907 0.645161 0.243243 2.718 ||| 1-1 ||| 23 31
```

The format can also a represent a tree-to-string rule, which has no Hiero equivalent:

```
all [NN][X] [NP] ||| &#47784;&#46304; [NN][X] [X] ||| 0.869565 0.627907 0.645161 0.243243 2.718 ||| 1-1 ||| 23 31
```

Usually, you will also need these 'glue' rules:

```
<s> [X][S] </s> [X] ||| <s> [X][S] </s> [TOP] ||| 1.0 ||| 1-1 ||| 5442.66 323.246
<s> [X][NP] </s> [X] ||| <s> [X][NP] </s> [TOP] ||| 1.0 ||| 1-1 ||| 5442.66 323.246
<s> [X] ||| <s> [S] ||| 1 |||
[X][S] </s> [X] ||| [X][S] </s> [S] ||| 1 ||| 0-0
[X][S] [X][X] [X] ||| [X][S] [X][X] [S] ||| 2.718 ||| 0-0 1-1
```

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 moses.ini -T trace-file ; cat trace-file
[...]
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 coveres 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 moses.ini -T trace-file ; cat trace-file
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 5 8
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 this can be the case 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.

2.3.4 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 an 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, but 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 are combined 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) and 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` is used to specify privileged non-terminals. These are used for unknown words (unless there is a unknown word label file) and to define the non-terminal label 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 considered can be limited with `rule-limit`. Default is 5000.

2.3.5 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, e.g. `-hierarchical and -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 create 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 creates 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 sub-phrases. 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 option leads to the count of one for each rule.
- `-NoNonTermFirstWord`: Disallows that a rule starts with a non-terminal.

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

- `-OnlyDirect`: only estimates direct conditional probabilities. Note that this option needs to be specified for both rule extraction and rule scoring.
- `-NoLex`: only includes rule-level conditional probabilities, not lexical scores.
- `-GoodTuring`: Uses 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> </tree>
```

So, constituents are surrounded by an 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"/> {\bf the cat}
```

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

```
____ NP _____
/     /   \     \
NNP    NNP    NNP    NNP
Prime Minister Ariel Sharon
```

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

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

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:

```
NP
/   \
/     \^NP
/     /    \
/     /      \^NP
/     /      /    \
NNP    NNP    NNP    NNP
Prime Minister Ariel Sharon
```

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:

- `-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) (usually 2)] [input text pt] [output directory]
```

eg.

```
~/CreateOnDiskPt 1 1 5 100 2 pt.txt pt.folder
```

The moses.ini file should also be changed so that the binary files is used instead of the text file. You should change it from:

```
[ttable-file]
6 0 0 5 pt.txt
```

to

```
[ttable-file]
2 0 0 5 pt.folder
```

2.4 Common Pitfalls

2.4.1 Using Meta-symbols in Non-terminal Symbols (e.g., CCG)

Often a syntactic formalism will use symbols that are part of the meta-symbols that denote non-terminal boundaries in the SCFG rule table, and glue grammar. For example, in Combinatory Categorial Grammar (CCG, Steedman, 2000), it is customary to denote grammatical features by placing them after the non-terminal symbol inside square brackets, as in S[dcl] (declarative sentence) vs. S[q] (interrogative sentence).

Although such annotations may be useful to discriminate good translations from bad, including square brackets in the non-terminal symbols themselves can confuse Moses. Some users have reported that category symbols were mangled (by splitting them at the square brackets) after converting to an on-disk representation (and potentially in other scenarios – this is currently an open issue). A way to side-step this issue is to escape square brackets with a symbol that is not part of the meta-language of the grammar files, e.g. using the underscore symbol:

S[dcl] => S_dcl_

and

S[q] => S_q_

before extracting a grammar. This should be done in all data or tables that mention such syntactic categories. If the rule table is automatically extracted, it suffices to escape the categories in the <tree label="..."...> mark-up that is supplied to the training script. If you roll your own rule tables (or use an ’unknown-lhs’ file), you should make sure they are properly escaped.

2.5 Different Kinds of Syntax Models

2.5.1 String-to-Tree

Most SCFG-based machine translation decoders at the current time are designed to uses hierarchical phrase-based grammar (Chiang, 2005) or syntactic grammar. Joshua, CDEC, Jane are some of the open-sourced systems that have such decoders.

The hierarchical phrase-based grammar is well described elsewhere so we will not go into details here. Briefly, the non-terminals are not labelled with any linguistically-motivated labels. By convention, non-terminals have been simply labelled as X, eg.

```
X --> der X1 ||| the X1
```

Usually, a set of glue rules are needed to ensure that the decoder always output an answer. By convention, the non-terminals for glue rules are labelled as S, eg.

```
S --> <s> ||| <s>
S --> X1 </s> ||| X1 </s>
S --> X1 X2 ||| X1 X2
```

In a syntactic model, non-terminals are labelled with linguistically-motivated labels such as ’NOUN’, ’VERB’ etc. For example:

```
DET --> der ||| the
ADJ --> kleines ||| small
```

These labels are typically obtained by parsing the target side of the training corpus. (However, it is also possible to use parses of the source side which has been projected onto the target side (Ambati and Chen, 2007)).

The input to the decoder when using this model is a conventional string, as in phrase-based and hierarchical phrase-based models. The output is a string. However, the CFG-tree derivation of the output (target) can also be obtained (in Moses by using the -T argument), the non-terminals in this tree will be labelled with the linguistically-motivated labels.

For these reasons, these syntactic models are called ’target’ syntax models, or ’string-to-tree’ model, by many in the Moses community and elsewhere. (Some papers by people at ISI inverted this naming convention due to their adherance to the noisy-channel framework).

The implementation of string-to-tree models is fairly standard and similar across different open-source decoders such as Moses, Joshua, cdec and Jane.

There is a ’string-to-tree’ model in sample model:

```
http://www.statmt.org/moses/download/sample-models.tgz
```

The input to the model is the string:

```
das ist ein kleines haus
```

The output string is

```
this is a small house
```

The target tree it produces is

```
(TOP <s> (S (NP this) (VP (V is) (NP (DT a) (ADJ small) (NN house)))) </s>)
```

RECAP - The input is a string, the output is a tree with linguistically-motivated labels.

2.5.2 Tree-to-string

Unlike the string-to-tree model, the tree-to-string model is not as standardized across different decoders. This section describes the Moses implementation.

Input tree representation

The input to the decoder is a parse tree, not a string. For Moses, the parse tree should be formatted using XML, described in

```
http://www.statmt.org/moses/?n=Moses.SyntaxTutorial (page \pageref{syntax-tutorial.tex})
Section Training Syntax Models
```

The decoder converts the parse tree into an annotated string (a chart?). Each span in the chart is labelled with the non-terminal from the parse tree. For example, the input

```
<tree label="NP"> <tree label="DET"> the </tree> <tree label="NN"> cat </tree> </tree>
```

is converted to an annotated string

```
the   cat
-DET- -NN--
----NP-----
```

To support easier glue rules, the non-terminal ’X’ is also added for every span in the annotated string. Therefore, the input above is actually converted to:

```
the   cat
-DET- -NN--
--X-- --X--
----NP-----
-----X-----
```

Translation rules

During decoding, the non-terminal of the rule that spans a substring in the sentence must match the label on the annoated string. For example, the following rules can be applied to the above sentence.

```
NP --> the katze ||| die katze
NP --> the NN1 ||| der NN1
NP --> DET1 cat ||| DET1 katze
NP --> DET1 NN2 ||| DET1 NN2
```

However, these rules can't as they don't match one or more non-terminals.

```
VB --> the katze ||| die katze
NP --> the ADJ1 ||| der ADJ1
NP --> ADJ1 cat ||| ADJ1 katze
ADV --> ADJ1 NN2 ||| ADJ1 NN2
```

Therefore, non-terminal in the translation rules in a tree-to-string model acts as constraints on which rules can be applied. This constraint is in addition to the usual role of non-terminals.

A feature which is currently unique to the Moses decoder is the ability to separate out these two roles. Each non-terminal in all translation rules is represented by two labels:

```
1. The 'source' non-terminal which constrains rules to the input parse tree
2. 'Target' non-terminal which has the normal parsing role.
```

When we need to differentiate source and target non-terminals, the translation rules are instead written like this:

```
NP --> the NN1 ||| X --> der X1
```

This rule indicates that the non-terminal should span a NN constituent in the input text, and that the whole rule should span an NP constituent. The target non-terminals in this rule are both X, therefore, this rule would be considered part of tree-to-string grammar.

(Using this notation is probably wrong as the source sentence is not properly parsed - see next section. It may be better to express the Moses tree-to-string grammar as a hierarchical grammar, with added constraints. For example:

```
X --> the X1 ||| der X1 ||| LHS = NP, X_1 = NN
```

However, this may be even more confusing so we will stick with our convention for now.)

RECAP - Grammar rules in Moses have 2 labels for each non-terminals; one to constrain the non-terminal to the input parse tree, the other is used in parsing.

2.5.3 Consequences

1. The Moses decoder always checks the source non-terminal, even when it is decoding with a string-to-string or string-to-tree grammar. For example, when checking whether the following rule can be applied

```
X --> der X1 ||| the X1
```

the decoder will check whether the RHS non-terminal, and the whole rule, spans an input parse constituent X. Therefore, even when decoding with a string-to-string or string-to-tree grammar, it is necessary to add the X non-terminal to every input span. For example, the input string `the cat` must be annotated as follows

```
the   cat
--X-- --X--
-----X-----
```

to allow the string to be decoded with a string-to-string or string-to-tree grammar.

2. There is no difference between a linguistically derived non-terminal label, such as NP, VP etc, and the non-linguistically motivated X label. They can both be used in one grammar, or even 1 translation rule. This 'mixed-syntax' model was explored in (Hoang and Koehn, 2010) and in Hieu's thesis[5]

3. The source non-terminals in translation rules are used just to constrain against the input parse tree, not for parsing. For example, if the input parse tree is

```
(VP (NP (PRO he)) (VB goes))
```

and tree-to-string rules are:

```
PRO --> he ||| X --> il
VB --> goes ||| X --> va
VP --> NP1 VB2 ||| X --> X1 X2
```

[5] http://www.statmt.org/~s0565741/ddd.pdf

This will create a valid translation. However, the span over the word 'he' will be labelled as PRO by the first rule, and NP by the 3rd rule. This is illustrated in more detail in Hieu's thesis Section 4.2.11.

4. To avoid the above and ensure that source spans are always consistently labelled, simply project the non-terminal label to both source and target. For example, change the rule

```
VP --> NP1 VB2 ||| X --> X1 X2
```

to

```
VP --> NP1 VB2 ||| VP --> NP1 VB2
```

2.5.4 Format of text rule table

The format of the Moses rule table is different from that used by Hiero, Joshua and cdec, and has often been a source of confusion. We shall attempt to explain the reasons in this section.

The format is derived from the Pharaoh/Moses phrase-based format. In this format, a translation rule

```
a b c --> d e f , with word alignments a1, a2 ..., and probabilities p1, p2, ...
```

is formatted as

```
a b c ||| d e f ||| p1 p2 ... ||| a1 a2 ...
```

For a hierarchical pb rule,

```
X --> a X1 b c X2 ||| d e f X2 X1
```

The Hiero/Joshua/cdec format is

```
X ||| a [X,1] b c [X,2] ||| d e f [X,2] [X,1] ||| p1 p2 ...
```

The Moses format is

```
a [X][X] b c [X][X] [X] ||| d e f [X][X] [X][X] [X] ||| p1 p2 ... ||| 1-3 4-4
```

For a string-to-tree rule,

```
VP --> a X1 b c X2 ||| d e f NP2 ADJ1
```

the Moses format is

```
a [X][ADJ] b c [X][NP] [X] ||| d e f [X][NP] [X][ADJ] [VP] ||| p1 p2 ... ||| 1-3 4-4
```

For a tree-to-string rule,

```
VP --> a ADJ1 b c NP2 ||| X --> d e f X2 X1
```

The Moses format is

```
a [ADJ][X] b c [NP][X] [VP] ||| d e f [NP][X] [ADJ][X] [X] ||| p1 p2 ... ||| 1-3 4-4
```

The main reasons for the difference between the Hiero/Joshua/cdec and Moses formats are as follows:

1. The text rule table should be easy to convert to a binary, on-disk format. We have seen in the community that this allows much large models to be used during decoding, even on memory limited servers. To make the conversion efficient, the text rule table must have the following properties:
   (a) For every rule, the sequence of terminals and non-terminals in the first column (the 'source' column) should match the lookup sequence that the decoder will perform.

(b) The file can be sorted so that the first column is in alphabetical order. The decoder needs to look up the target non-terminals on the RHS of each rule so the first column consist of source terminals and non-terminal, AND target non-terminals from the RHS.

2. The phrase probability calculations should be performed efficiently. To calculate p(t | s) = count(t,s) / count(s) the extract file must be sorted in contiguous order so that each count can be performed and used to calculate the probability, then discarded immediately to save memory. Similarly for p(s | t) = count(t,s) / count(t)

The Hiero/Joshua/cdec file format is sufficient for hierarchical models, but not for the various syntax models supported by Moses.

Subsection last modified on June 18, 2012, at 04:28 PM

2.6 Baseline System

NB: The build system was updated on 2012-05-31. This documentation will be updated to match shortly

History:
April 16th, 2012: Add EMS (Barry Haddow)
April 13th, 2012: Created (Barry Haddow)

2.6.1 Overview

This guide assumes that you have successfully installed (page **??**) Moses, and would like to see how to use parallel data to build a real phrase-based translation system. The process requires some familiarity with UNIX and, ideally, access to a Linux server. It can be run on a laptop, but could take about a day and requires at least 2G of RAM, and about 20G of free disk space (these requirements are just educated guesses, so if you have a different experience then please mail support[6]).

If you want to save the effort of typing in all the commands on this page (and see how the pros manage their experiments), then skip straight to the experiment management system (page **??**) instructions below. But I'd recommend that you follow through the process manually, at least once, just to see how it all works.

2.6.2 Installation

The minimum software requirements are:

- Moses (obviously!)
- GIZA++[7], for word-aligning your parallel corpus

[6] http://www.statmt.org/moses/?n=Moses.MailingLists

[7] http://code.google.com/p/giza-pp/

- Either IRSTLM[8] or SRILM[9] for Language model estimation

This guide will use IRSTLM, since it's LGPL licensed (like Moses) and therefore available for commercial use. The Moses tool-chain defaults to SRILM, but it requires an expensive licence for non-academic use.

For the purposes of this guide, I will assume that you're going to install all the tools and data in your home directory (i.e. ~/), and that you've already downloaded and compiled Moses into ~/mosesdecoder. And you're going to run Moses from there.

Installing GIZA++

GIZA++ is hosted at Google Code[10], and a mirror of the original documentation can be found here[11]. I recommend that you download the latest version from Google Code - I'm using 1.0.7 in this guide so I downloaded and built it with the following commands (issued in my home directory):

```
wget http://giza-pp.googlecode.com/files/giza-pp-v1.0.7.tar.gz
tar xzvf giza-pp-v1.0.7.tar.gz
cd giza-pp
make
```

This should create the binaries ~/giza-pp/GIZA++-v2/GIZA++, ~/giza-pp/GIZA++-v2/snt2cooc.out and ~/giza-pp/mkcls-v2/mkcls. These need to be copied to somewhere that Moses can find them as follows

```
cd ~
mkdir external-bin-dir
cp ~/giza-pp/GIZA++-v2/GIZA++ ~/giza-pp/GIZA++-v2/snt2cooc.out ~/giza-pp/mkcls-v2/mkcls external-bin-dir
```

When you are running the training, you need to tell the training script where GIZA++ was installed with the argument.

```
train-model.perl -external-bin-dir $HOME/external-bin-dir
```

Installing IRSTLM

[8] http://hlt.fbk.eu/en/irstlm

[9] http://www.speech.sri.com/projects/srilm/download.html

[10] http://giza-pp.googlecode.com/

[11] http://www.statmt.org/moses/giza/GIZA++.html

IRSTLM is a language modelling toolkit from FBK, and is hosted on sourceforge[12]. Again, you should download the latest version. I used version 5.70.04 for this guide so assuming you downloaded the tarball into your home directory (and making the obvious changes if you download a later version) the following commands should build and install IRSTLM:

```
tar zxvf irstlm-5.70.04.tgz
cd irstlm-5.70.04
./regenerate-makefiles.sh
./configure --prefix=$HOME/irstlm
make
make install
```

You should now have several binaries and scripts in `~/irstlm/bin`, in particular `build-lm.sh`

2.6.3 Corpus Preparation

To train a translation system we need parallel data (text translated into two different languages) which is aligned at the sentence level. Luckily there's plenty of this data freely available, and for this system I'mm going to use a small (only 130,000 sentences!) data set released for the 2012 Workshop in Machine Translation. To get the data we want, we have to download a fairly large (700MB) tarball and unpack it (into a corpus directory in our home directory) as follows

```
cd
mkdir corpus
cd corpus
wget http://www.statmt.org/wmt12/training-parallel.tgz
tar zxvf training-parallel.tgz
```

If you look in the `~/corpus/training` directory you'll see that there's data from europarl (european parliamentary proceedings) and news-commentary (news analysis from project syndicate) in various languages. We're going to build a French-English (fr-en) translation system using the news commentary data set, but feel free to use one of the other language pairs if you prefer.

To prepare the data for training the translation system, we have to perform the following steps:

- **tokenisation**: This means that spaces have to be inserted between (e.g.) words and punctuation.
- **truecasing**: The initial words in each sentence are converted to their most probable casing. This helps reduce data sparsity.
- **cleaning**: Long sentences and empty sentences are removed as they can cause problems with the training pipeline, and obviously mis-aligned sentences are removed.

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

The tokenisation can be run as follows:

```
~/mosesdecoder/scripts/tokenizer/tokenizer.perl -l en < ~/corpus/training/news-commentary-v7.fr-en.en  \
> ~/corpus/news-commentary-v7.fr-en.tok.en
~/mosesdecoder/scripts/tokenizer/tokenizer.perl -l fr < ~/corpus/training/news-commentary-v7.fr-en.fr  \
> ~/corpus/news-commentary-v7.fr-en.tok.fr
```

The truecaser first requires training, in order to extract some statistics about the text:

```
~/mosesdecoder/scripts/recaser/train-truecaser.perl --model ~/corpus/truecase-model.en --corpus  \
~/corpus/news-commentary-v7.fr-en.tok.en
~/mosesdecoder/scripts/recaser/train-truecaser.perl --model ~/corpus/truecase-model.fr --corpus  \
~/corpus/news-commentary-v7.fr-en.tok.fr
```

Truecasing uses another script from the Moses distribution:

```
~/mosesdecoder/scripts/recaser/truecase.perl --model ~/corpus/truecase-model.en < ~/corpus/news-commentary-v7.fr-en.tok.en  \
> ~/corpus/news-commentary-v7.fr-en.true.en
~/mosesdecoder/scripts/recaser/truecase.perl --model ~/corpus/truecase-model.fr < ~/corpus/news-commentary-v7.fr-en.tok.fr  \
> ~/corpus/news-commentary-v7.fr-en.true.fr
```

Finally we clean, limiting sentence length to 80:

```
~/mosesdecoder/scripts/training/clean-corpus-n.perl ~/corpus/news-commentary-v7.fr-en.true fr en  \
~/corpus/news-commentary-v7.fr-en.clean 1 80
```

Notice that the last command processes both sides at once.

2.6.4 Language Model Training

The language model (LM) is used to ensure fluent output, so it is built with the target language (i.e English). The IRSTLM documentation gives a full explanation of the command-line options, but the following will build an appropriate 3-gram language model, removing singletons, smoothing with improved Kneser-Ney, and adding sentence boundary symbols:

```
mkdir ~/lm
cd ~/lm
~/irstlm/bin/add-start-end.sh  < ~/corpus/news-commentary-v7.fr-en.true.en > news-commentary-v7.fr-en.sb.en
export IRSTLM=$HOME/irstlm; ~/irstlm/bin/build-lm.sh -i  news-commentary-v7.fr-en.sb.en  -t ./tmp  -p  \
-s improved-kneser-ney -o news-commentary-v7.fr-en.lm.en
~/irstlm/bin/compile-lm --text yes news-commentary-v7.fr-en.lm.en.gz news-commentary-v7.fr-en.arpa.en
```

This should give a language model in the `*.arpa.en` file, which we'll then binarise (for faster loading) using KenLM

```
~/mosesdecoder/bin/build_binary  news-commentary-v7.fr-en.arpa.en  news-commentary-v7.fr-en.blm.en
```

You can check the language model by querying it, e.g.

```
[bhaddow@hoenir lm]$ echo "is this an English sentence ?" | ~/mosesdecoder/bin/query news-commentary-v7.fr-en.blm.en
Loading statistics:
user    9.99e-07
sys     9.99e-07
rss    27368 kB
is=64 2 -2.53708  this=41 3 -1.10739  an=44 2 -3.4964 English=14683 1 -5.23585 sentence=3118 2 -2.61447 ?=7 1 -3.24089  </s>=2 2 -0.029675
Total: -18.2618 OOV: 0
After queries:
user    1.999e-06
sys     9.99e-07
rss    27368 kB
Total time including destruction:
user    1.999e-06
sys     9.99e-07
rss     1472 kB
```

2.6.5 Training the Translation System

Finally we come to the main event - training the translation model. This will run word-alignment (using GIZA++), phrase extraction and scoring, create lexicalised reordering tables and create your Moses configuration file, all with a single command. I recommend that you create an appropriate directory as follows, and then run the training command, catching logs:

```
mkdir ~/working
cd ~/working
nohup nice ~/mosesdecoder/scripts/training/train-model.perl  -root-dir train -corpus ~/corpus/news-commentary-v7.fr-en.clean \
-f fr -e en -alignment grow-diag-final-and -reordering msd-bidirectional-fe  \
-lm 0:3:$HOME/lm/news-commentary-v7.fr-en.blm.en:8 >& training.out &
```

This took about 2 hours on an Intel Xeon X5550, with approx 120G Ram. Once it's finished there should be a `moses.ini` file in the directory `~/working/train/model`. You can use this ini file to decode, but there's a couple of problems with it. The first is that it's very slow to load, but we can fix that by *binarising* the phrase table and reordering table, i.e. compiling them into a format that can be load quickly. The second problem is that the weights used by Moses to weight the different models against each other are not optimised - if you look at the `moses.ini` file you'll see that they're set to default values like 0.2, 0.3 etc. To find better weights we need to *tune* the translation system, which leads us on to the next step...

2.6.6 Tuning

This is the slowest part of the process, so you might want to line up something to read whilst it's progressing. Tuning requires a small amount of parallel data, separate from the training data, so again we'll download some data kindly provided by WMT. Run the following commands (from your home directory again) to download the data and put it in a sensible place.

```
cd ~/corpus
wget http://www.statmt.org/wmt12/dev.tgz
tar zxvf dev.tgz
```

We're going to use news-test2008 for tuning, so we have to tokenise and truecase it first (don't forget to use the correct language if you're not building a fr->en system)

```
cd ~/corpus
~/mosesdecoder/scripts/tokenizer/tokenizer.perl -l en < dev/news-test2008.en > news-test2008.tok.en
~/mosesdecoder/scripts/tokenizer/tokenizer.perl -l fr < dev/news-test2008.fr > news-test2008.tok.fr
~/mosesdecoder/scripts/recaser/truecase.perl --model truecase-model.en < news-test2008.tok.en > news-test2008.true.en
~/mosesdecoder/scripts/recaser/truecase.perl --model truecase-model.fr < news-test2008.tok.fr > news-test2008.true.fr
```

Now go back to the directory we used for training, and launch the tuning process:

```
cd ~/working
nohup nice ~/mosesdecoder/scripts/training/mert-moses.pl ~/corpus/news-test2008.true.fr ~/corpus/news-test2008.true.en \
~/mosesdecoder/bin/moses  train/model/moses.ini --mertdir ~/mosesdecoder/bin/ &> mert.out &
```

If you have several cores at your disposal, then it'll be a lot faster to run Moses multi-threaded. Add `-decoder-flags="-threads 4"` to the last line above in order to run the decoder with 4 threads. With this setting, tuning took about 4 hours for me.

The end result of tuning is an ini file with trained weights, which should be in `~/working/mert-work/moses.ini` if you've used the same directory structure as me.

2.6.7 Testing

You can now run Moses with

```
~/mosesdecoder/bin/moses -f ~/working/mert-work/moses.ini
```

and type in your favourite French sentence to see the results. You'll notice, though, that the decoder takes at least a couple of minutes to start-up. In order to make it start quickly, we can binarise the phrase-table and lexicalised reordering models. To do this, create a suitable directory and binarise the models as follows:

```
mkdir ~/working/binarised-model
cd ~/working
~/mosesdecoder/bin/processPhraseTable -ttable 0 0 train/model/phrase-table.gz -nscores 5 -out binarised-model/phrase-table
~/mosesdecoder/bin/processLexicalTable -in train/model/reordering-table.wbe-msd-bidirectional-fe.gz -out binarised-model/reordering-table
```

Then make a copy of the `~/working/mert-work/moses.ini` in the binarised-model directory and change the phrase and reordering tables to point to the binarised versions. My diff between the tuned and binarised versions is as follows:

```
[bhaddow@hoenir working]$ diff mert-work/moses.ini binarised-model/moses.ini
22c22
< 0 0 0 5 /home/bhaddow/working/train/model/phrase-table.gz
---
> 1 0 0 5 /home/bhaddow/working/binarised-model/phrase-table
38c38
< 0-0 wbe-msd-bidirectional-fe-allff 6 /home/bhaddow/working/train/model/reordering-table.wbe-msd-bidirectional-fe.gz
---
> 0-0 wbe-msd-bidirectional-fe-allff 6 /home/bhaddow/working/binarised-model/reordering-table
```

Loading and running a translation is pretty fast:

```
[bhaddow@hoenir working]$ echo "faire revenir les militants sur le terrain et convaincre que le vote est utile ." |  \
~/mosesdecoder/bin/moses -f binarised-model/moses.ini
Defined parameters (per moses.ini or switch):
config: binarised-model/moses.ini
distortion-file: 0-0 wbe-msd-bidirectional-fe-allff 6 /home/bhaddow/working/binarised-model/reordering-table
distortion-limit: 6
input-factors: 0
lmodel-file: 8 0 3 /home/bhaddow/lm/news-commentary-v7.fr-en.blm.en
mapping: 0 T 0
threads: 4
ttable-file: 1 0 0 5 /home/bhaddow/working/binarised-model/phrase-table
ttable-limit: 20
weight-d: 0.0810812 0.0919838 0.0224705 0.0238237 0.128651 0.0299205 0.0642029
weight-l: 0.0822222
weight-t: 0.0411399 0.076291 0.0936291 0.0878611 0.0430955
weight-w: -0.133628
Loading lexical distortion models...have 1 models
```

```
Creating lexical reordering...
weights: 0.092 0.022 0.024 0.129 0.030 0.064
binary file loaded, default OFF_T: -1
Start loading LanguageModel /home/bhaddow/lm/news-commentary-v7.fr-en.blm.en : [0.000] seconds
Finished loading LanguageModels : [0.000] seconds
Start loading PhraseTable /home/bhaddow/working/binarised-model/phrase-table : [0.000] seconds
filePath: /home/bhaddow/working/binarised-model/phrase-table
Finished loading phrase tables : [0.000] seconds
IO from STDOUT/STDIN
Created input-output object : [0.000] seconds
Translating line 0  in thread id 140332796847888
Translating: faire revenir les militants sur le terrain et convaincre que le vote est utile .

reading bin ttable
size of OFF_T 8
binary phrasefile loaded, default OFF_T: -1
binary file loaded, default OFF_T: -1
Collecting options took 0.350 seconds
Search took 0.680 seconds
Bring militants on the ground and convince that the vote is useful .
BEST TRANSLATION: Bring militants on the ground and convince that the vote is useful . [111111111111111]  [total=-8.467] <<0.000,
-13.000, 0.000, -3.256, 0.000, 0.000, -3.163,  0.000, 0.000, -81.169, -4.316, -13.801, -5.485, -15.726, 6.999>>
Translation took 0.690 seconds
Finished translating
```

The translation is quite rough, but understandable - bear in mind this is a very small data set for general domain translation. Also note that your results may differ slightly due to non-determinism in the tuning process.

At this stage, your probably wondering how good the translation system is. To measure this, we use another parallel data set (the test set) distinct from the ones we've used so far. Let's pick newstest2011, and so first we have to tokenise and truecase it as before

```
cd ~/corpus
~/mosesdecoder/scripts/tokenizer/tokenizer.perl -l en < dev/newstest2011.en > newstest2011.tok.en
~/mosesdecoder/scripts/tokenizer/tokenizer.perl -l fr < dev/newstest2011.fr > newstest2011.tok.fr
~/mosesdecoder/scripts/recaser/truecase.perl --model truecase-model.en < newstest2011.tok.en > newstest2011.true.en
~/mosesdecoder/scripts/recaser/truecase.perl --model truecase-model.fr < newstest2011.tok.fr > newstest2011.true.fr
```

The model that we've trained can then be *filtered* for this test set, meaning that we only retain the entries needed translate the test set. This will make the translation a lot faster.

```
cd ~/working
~/mosesdecoder/scripts/training/filter-model-given-input.pl filtered-newstest2011 mert-work/moses.ini ~/corpus/newstest2011.true.fr  \
-Binarizer ~/mosesdecoder/bin/processPhraseTable
```

You can test the decoder by first translating the test set (takes a wee while) then running the bleu script on it:

```
nohup nice ~/mosesdecoder/bin/moses -f ~/working/filtered-newstest2011/moses.ini < ~/corpus/newstest2011.true.fr \
> ~/working/newstest2011.translated.en 2> ~/working/newstest2011.out &
~/mosesdecoder/scripts/generic/multi-bleu.perl -lc ~/corpus/newstest2011.true.en < ~/working/newstest2011.translated.en
```

This gives me a bleu score of 23.2 (in comparison, the best result at wmt11 was 30.5[13], although it should be cautioned that this uses NIST bleu, which does its own tokenisation, so there will be 1-2 points difference in the score anyway)

2.6.8 Experiment Management System (EMS)

If you've been through the effort of typing in all the commands, then by now you're probably wondering if there's an easierway. If you've skipped straight down here without bothering about the manual route then, well, you may have missed on a useful Moses "rite of passage".

The easier way is, of course, to use the EMS (page **??**). To use EMS, you'll have to install a few dependencies, as detailed on the EMS page, and then you'll need this config[14] file. Make a directory `~/working/experiments` and place the config file in there. If you open it up, you'll see the `home-dir` variable defined at the top - then make the obvious change. If you set the home directory, download the train, tune and test data and place it in the locations described above, then this config file should work. The only wrinkle is that EMS currently (April 2012) does not support training with IRSTLM, so either you'll have to install and use SRILM (and refer to the main [EMS -> FactoredTraining.EMS]] documentation to see how to set up training) or see above (page **??**) for instructions on how to train the LM with IRSTLM (if you haven't already done so).

To run EMS from the `experiments` directory, you can use the command:

```
nohup nice ~/mosesdecoder/scripts/ems/experiment.perl -config config -exec &> log &
```

then sit back and wait for the bleu score to appear in `evaluation/report.1`

Subsection last modified on November 23, 2012, at 12:41 AM

[13] http://matrix.statmt.org/matrix/systems_list/1669

[14] http://www.statmt.org/moses/uploads/Moses/config

3

User Guide

3.1 Support Tools

3.1.1 Overview

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

The following basic tools are described elsewhere:

- Moses decoder (page **??**)
- Training script `train-model.perl` (page **??**)
- Corpus preparation `clean-corpus-n.perl` (page **??**)
- Minimum error rate training (tuning) [[`mert-moses.pl` (page **??**)

3.1.2 Content

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

3.1.3 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
```

3.1.4 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

3.1.5 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-model-given-input.pl filter-dir config input-file -Binarizer binarizer
```

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

3.1.6 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
```

3.1.7 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[1]. It requires that text is wrapped into a SGML format. This format is used for instance by the NIST evaluation[2] and the WMT Shared Task evaluations[3]. See the latter for more details on using this script.

[1] http://www.nist.gov/speech/tests/mt/2008/scoring.html

[2] http://www.nist.gov/speech/tests/mt/2009/

[3] http://www.statmt.org/wmt09/translation-task.html

3.1.8 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 ...
```

3.1.9 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
```

`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.

3.1.10 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.)

3.1.11 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
```

3.1.12 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 recasing model in the directory MODEL. Since the script needs to run SRILM's ngram-count and the Moses train-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 SGML] > OUT
```

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.

3.1.13 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.

3.1.14 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[4] 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.

Subsection last modified on April 25, 2012, at 07:54 PM

3.2 External Tools

A very active community is engaged in statistical machine translation research, which has produced a number of tools that may be useful for training a Moses system. Also, the more linguistically motivated models (factored model, syntax model) require tools to the linguistic annotation of corpora.

In this section, we list some useful tools. If you know (or are the developer of) anything we missed here, please contact us and we can add it to the list. For more comprehensive listings of MT tools, refer to the following pages:

- List of Free/Open-source MT Tools[5], maintained by Mikel Forcada.
- TAUS Tracker[6], a comprehensive list of Translation and Language Technology tools maintained by TAUS.

3.2.1 Contents

- Berkeley Word Aligner (page **??**)
- Multi-threaded GIZA++ (page **??**)
- Evaluation Metrics (page **??**)
- Translation Error Rate (TER) (page **??**)
- METEOR (page **??**)

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

[5] http://www.fosmt.org

[6] http://www.taustracker.com/

- Part-of-Speech Taggers (page **??**)
- MXPOST (English) (page **??**)
- TreeTagger (English, French, Spanish, German, Italian, Dutch, Bulgarian, Greek) (page **??**)
- Syntactic Parsers (page **??**)
- Collins (English) (page **??**)
- BitPar (German, English) (page **??**)
- LoPar (German) (page **??**)
- Berkeley Parser (page **??**)
- Other Open Source Machine Translation Systems (page **??**)
- Joshua (page **??**)
- cdec (page **??**)
- Apertium (page **??**)
- Docent (page **??**)

3.2.2 Berkeley Word Aligner

The BerkeleyAligner[7] (Sourceforge[8]) is a word alignment software package that implements recent innovations in unsupervised word alignment. It is implemented in Java.

Installation:

```
mkdir /my/installation/dir
cd /my/installation/dir
wget http://berkeleyaligner.googlecode.com/files/berkeleyaligner_unsupervised-2.1.tar.gz
tar xzf berkeleyaligner_unsupervised-2.1.tar.gz
```

Test:

```
cd berkeleyaligner
chmod +x align
./align example.conf
```

[7] http://nlp.cs.berkeley.edu/Main.html#WordAligner

[8] http://code.google.com/p/berkeleyaligner/

3.2.3 Multi-threaded GIZA++

MGIZA[9] was developed by Qin Gao. It is an implementation of the popular GIZA++ word alignment toolkit to run multi-threaded on multi-core machines. Check the web site for more recent versions.

Installation:

```
mkdir /my/installation/dir
cd /my/installation/dir
wget http://www.cs.cmu.edu/~qing/release/mgiza-0.6.3-10-01-11.tar.gz
wget http://www.cs.cmu.edu/~qing/release/merge_alignment.py
```

The MGIZA binary and the script `merge_alignment.py` need to be copied in you binary directory that contains `GIZA++` (also note the name change for the binary).

```
cp src/mgiza BINDIR/mgizapp
cp scripts/merge_alignment.py BINDIR
```

MGIZA works with the training script `train-model.perl`. You indicate its use (opposed to regular GIZA++) with the switch `-mgiza`. The switch `-mgiza-cpus NUMBER` allows you to specify the number of CPUs.

3.2.4 Evaluation Metrics

Translation Error Rate (TER)

Translation Error Rate[10] is an error metric for machine translation that measures the number of edits required to change a system output into one of the references. It is implemented in Java.

Installation:

```
mkdir /my/installation/dir
cd /my/installation/dir
wget http://www.cs.umd.edu/~snover/tercom/tercom-0.7.25.tgz
tar xzf tercom-0.7.25.tgz
```

[9] http://geek.kyloo.net/software/doku.php

[10] http://www.cs.umd.edu/~snover/tercom/

METEOR

METEOR[11] is a metric that includes stemmed and synonym matches when measuring the similarity between system output and human reference translations.

Installation:

```
mkdir /my/installation/dir
cd /my/installation/dir
wget http://www.cs.cmu.edu/~alavie/METEOR/install-meteor-1.0.sh
sh install-meteor-1.0.sh
```

3.2.5 Part-of-Speech Taggers

MXPOST (English)

MXPOST was developed by Adwait Ratnaparkhi as part of his PhD thesis. It is a Java implementation of a maximum entropy model. It can be trained for any language pair for with annotated POS data exists.

Installation:

```
mkdir /your/installation/dir
cd /your/installation/dir
wget ftp://ftp.cis.upenn.edu/pub/adwait/jmx/jmx.tar.gz
tar xzf jmx.tar.gz
echo '#!/bin/ksh' > mxpost
echo 'export CLASSPATH=/your/installation/dir/mxpost.jar' >> mxpost
echo 'java -mx30m tagger.TestTagger /your/installation/dir/tagger.project' >> mxpost
```

Test:

```
echo 'This is a test .' | ./mxpost
```

The script `script/training/wrappers/make-factor-en-pos.mxpost.perl` is a wrapper script to create factors for a factored translation model. You have to adapt the definition of `$MXPOST` to point to your installation directory.

TreeTagger (English, French, Spanish, German, Italian, Dutch, Bulgarian, Greek)

[11] http://www.cs.cmu.edu/~alavie/METEOR/

TreeTagger[12] is a tool for annotating text with part-of-speech and lemma information.

Installation (Linux, check web site for other platforms):

```
mkdir /my/installation/dir
cd /my/installation/dir
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/tree-tagger-linux-3.2.tar.gz
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/tagger-scripts.tar.gz
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/install-tagger.sh
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/english-par-linux-3.1.bin.gz
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/french-par-linux-3.2-utf8.bin.gz
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/spanish-par-linux-3.1.bin.gz
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/german-par-linux-3.2.bin.gz
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/italian-par-linux-3.2-utf8.bin.gz
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/dutch-par-linux-3.1.bin.gz
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/bulgarian-par-linux-3.1.bin.gz
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/greek-par-linux-3.2.bin.gz
sh install-tagger.sh
```

The wrapper script `scripts/training/wrapper/make-pos.tree-tagger.perl` creates part-of-speech factors using TreeTagger in the format expected by Moses. The command has the required parameters `-tree-tagger DIR` to specify the location of your installation and `-l LANGUAGE` to specify the two-letter code for the language (`de`, `fr`, ...). Optional parameters are `-basic` to output only basic part-of-speech tags (`VER` instead of `VER:simp` – not available for all languages), and `-stem` to output stems instead of part-of-speech tags.

Treetagger can also shallow parse the sentence, labelling it with chunk tags. See their website[13] for details.

3.2.6 Syntactic Parsers

Collins (English)

Michael Collins[14] developed the first statistical parser as part of his PhD thesis. It is implemented in C.

Installation:

```
mkdir /your/installation/dir
cd /your/installation/dir
wget http://people.csail.mit.edu/mcollins/PARSER.tar.gz
```

[12] http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/

[13] http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/DecisionTreeTagger.html

[14] http://people.csail.mit.edu/mcollins/

```
tar xzf PARSER.tar.gz
cd COLLINS-PARSER/code
make
```

Collins parser also requires the installation of MXPOST (page **??**). A wrapper file to generate parse trees in the format required to train syntax models with Moses is provided in `scrips/training/wrapper/parse-en-collins.perl`.

BitPar (German, English)

Helmut Schmid developed BitPar[15], a parser for highly ambiguous probabilistic context-free grammars (such as treebank grammars). BitPar uses bit-vector operations to speed up the basic parsing operations by parallelization. It is implemented in C.

Installation:

```
mkdir /your/installation/dir
cd /your/installation/dir
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/BitPar/BitPar.tar.gz
tar xzf BitPar.tar.gz
cd BitPar/src
make
cd ../..
```

You will also need the parsing model for German which was trained on the Tiger treebank:

```
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/BitPar/GermanParser.tar.gz
tar xzf GermanParser.tar.gz
cd GermanParser/src
make
cd ../..
```

There is also an English parsing model.

LoPar (German)

LoPar[16] is an implementation of a parser for head-lexicalized probabilistic context-free grammars, which can be also used for morphological analysis.

Installation:

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

```
mkdir /my/installation/dir
cd /my/installation/dir
wget ftp://ftp.ims.uni-stuttgart.de/pub/corpora/LoPar/lopar-3.0.linux.tar.gz
tar xzf lopar-3.0.linux.tar.gz
cd LoPar-3.0
```

Berkeley Parser

Models provided for English, Bugarian, Arabic, Chinese, French, German.

http://code.google.com/p/berkeleyparser/

3.2.7 Other Open Source Machine Translation Systems

Joshua

Joshua[17] is a machine translation decoder for hierarchical models. Joshua development is centered at the Center for Language and Speech Processing at the Johns Hopkins University in Baltimore, Maryland. It is implemented in Java.

cdec

Cdec[18] is a decoder, aligner, and learning framework for statistical machine translation and other structured prediction models written by Chris Dyer in the University of Maryland Department of Linguistics. It is written in C++.

Apertium

Apertium[19] is an open source rule-based machine translation (RBMT) system, maintained principally by the University of Alicante and Prompsit Engineering.

Docent

Docent[20] is a decoder for phrase-based SMT that treats complete documents, rather than single sentences, as translation units and permits the inclusion of features with cross-sentence dependencies. It is developed by Christian Hardmeier and implemented in C++

Subsection last modified on October 30, 2012, at 09:13 PM

3.3 Advanced Features of the Decoder

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

[17] http://joshua.sourceforge.net/Joshua/Welcome.html

[18] http://cdec-decoder.org/

[19] http://www.apertium.org/

[20] https://github.com/chardmeier/docent/wiki

3.3.1 Content

- Lexicalized Reordering Models (page ??)
- Enhanced orientation detection (page ??)
- Binary Phrase Tables with On-demand Loading (page ??)
- Binary Reordering Tables with On-demand Loading (page ??)
- Compact Phrase Table (page ??)
- Compact Lexical Reordering Table (page ??)
- XML Markup (page ??)
- Generating n-Best Lists (page ??)
- Word-to-word alignment (page ??)
- Minimum Bayes Risk Decoding (page ??)
- Lattice MBR and Consensus Decoding (page ??)
- Handling Unknown Words (page ??)
- Output Search Graph (page ??)
- Early Discarding of Hypotheses (page ??)
- Maintaining stack diversity (page ??)
- Cube Pruning (page ??)
- Specifying Reordering Constraints (page ??)
- Multiple Translation Tables and Backoff Models (page ??)
- Pruning the Translation Table (page ??)
- Build Instructions (page ??)
- Usage Instructions (page ??)
- Pruning the Phrase Table based on Relative Entropy (page ??)
- Build Instructions (page ??)
- Usage Instructions (page ??)
- Parallelization (page ??)
- Multi-threaded Moses (page ??)
- Moses Server (page ??)
- Using Multiple Translation Systems in the Same Server (page ??)
- Amazon EC2 cloud (page ??)
- k-best batch MIRA Tuning (page ??)
- ZMERT Tuning (page ??)
- Lattice MERT (page ??)
- Continue Partial Translation (page ??)

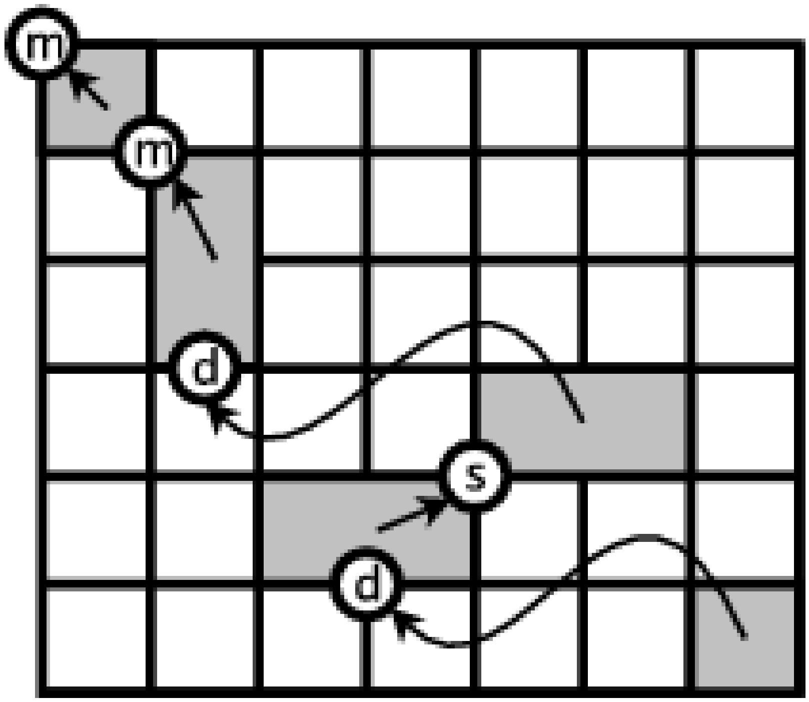
- Global Lexicon Model (page **??**)
- Incremental Training (page **??**)
- Introduction (page **??**)
- Initial Training (page **??**)
- Updates (page **??**)
- Distributed Language Model (page **??**)
- Installing and Compiling (page **??**)
- Configuration (page **??**)
- Example (page **??**)
- Suffix Arrays for Hierarchical Models (page **??**)
- Using the EMS (page **??**)
- Fuzzy Match Rule Table for Hierachical Models (page **??**)
- Translation Model Combination (page **??**)
- Linear Interpolation and Instance Weighting (page **??**)
- Fill-up (page **??**)

3.3.2 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.



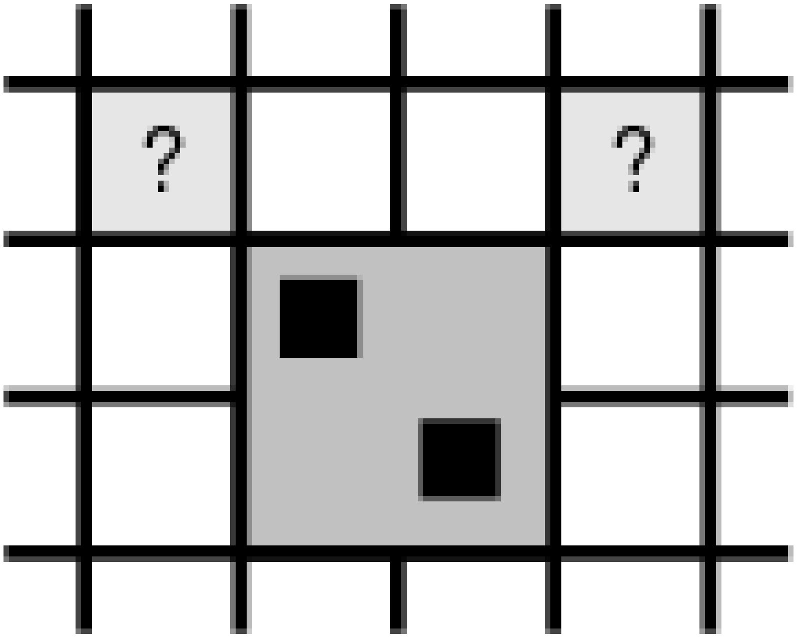
To put it more formally, we want to introduce a reordering model p_o that predicts an orientation type $\{m,s,d\}$ given the phrase pair currently used in translation:

orientation ϵ $\{m, s, d\}$

$p_o(orientation|f,e)$

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:

$p_o(orientation|f,e) = count(orientation,e,f) / \Sigma_o\ count(o,e,f)$

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:

$p_o(orientation) = \Sigma_f\ \Sigma_e\ count(orientation,e,f) / \Sigma_o\ \Sigma_f\ \Sigma_e\ count(o,e,f)$

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

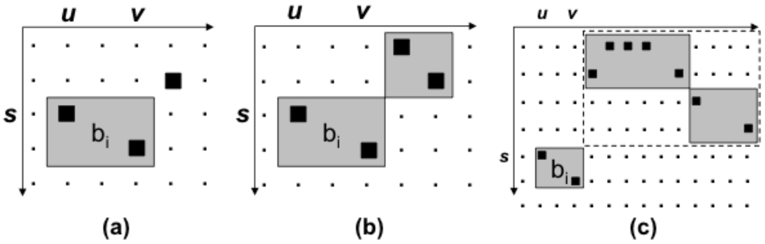
There are a number of variations of this lexicalized reordering model based on orientation types:

- **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. `bidrectional-monotonicity-f`). See more on training these models in the training section of this manual.

Enhanced orientation detection

As explained above, statistics about the orientation of each phrase can be collected by looking at the word alignment matrix, in particular by checking the presence of a word at the top left and right corners. This simple approach is capable of detecting a swap with a previous phrase that contains a word exactly aligned on the top right corner, see case (a) in the figure below. However, this approach cannot detect a swap with a phrase that does not contain a word with such an alignment, like the case (b). A variation to the way phrase orientation statistics are collected is the so-called **phrase-based orientation model** by Tillmann (2004)[21], which uses phrases both at training and decoding time. With the phrase-based orientation model, the case (b) is properly detected and counted during training as a swap. A further improvement of this method is the **hierarchical orientation model** by Galley and Manning (2008)[22], which is able to detect swaps or monotone arrangements between blocks even larger than the length limit imposed to phrases during training, and larger than the phrases actually used during decoding. For instance, it can detect at decoding time the swap of blocks in the case (c) shown below.



(Figure from Galley and Manning, 2008)

Empirically, the enhanced orientation methods should be used with language pairs involving significant word re-ordering.

3.3.3 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.

This section describes binary PT for phrase-based Moses only. For moses_chart see "On-Disk Rule Table"[23] in Syntax Tutorial.

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):

[21] http://www.aclweb.org/anthology-new/N/N04/N04-4026.pdf

[22] http://www.aclweb.org/anthology/D/D08/D08-1089.pdf

[23] http://www.statmt.org/moses/?n=Moses.SyntaxTutorial#OnDisk

```
export LC_ALL=C
cat phrase-table | sort | bin/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: If your data is encoded in UTF8, make sure you set the environment variable with the `export LC_ALL=C` before 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 only the filename stem `phrase-table` as phrase table and set the type to 1, i.e.:

```
[ttable-file]
1 0 0 5 phrase-table
```

Word-to-word alignment:

There are 2 arguments to the decoder that enables it to print out the word alignment information

```
-alignment-output-file [file]
```

print out the word alignment for the best translation to a file.

```
-print-alignment-info-in-n-best
```

print the word alignment information of each entry in the n-best list as an extra column in the n-best file.

Word alignment is included in the phrase-table by default (as of November 2012). To exclude them, add

```
--NoWordAlignment
```

as an argument to the score program.

When binarizing the phrase-table, the word alignment is also included by default. To turn this behaviour off for the phrase-based binarizer:

```
processPhraseTable -no-alignment-info ....
```

Or

```
processPhraseTableMin -no-alignment-info ....
```

(For the compact phrase-table representation).

There is no way to exclude word alignment information from the chart-based binarization process.

Phrase-based binary format When word alignment information is stored, the two output files ".srctree" and " .tgtdata" will end with the suffix ".wa".

Note: The argument

```
-use-alignment-info
-print-alignment-info
```

has been deleted from the decoder. `-print-alignment-info` did nothing. `-use-alignment-info` is now inferred from the arguments

```
-alignment-output-file
-print-alignment-info-in-n-best
```

Additionally, the

```
-include-alignment-in-n-best
```

has been renamed

```
-include-segmentation-in-n-best
```

to reflect what it actually does.

The word alignment MUST be enabled during bianrization, otherwise the decoder will

```
1. complain
2. carry on blindly but doesn't print any word alignment
```

3.3.4 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.

3.3.5 Compact Phrase Table

A compact phrase table implementation is available that is around 6 to 7 times smaller and than the original binary phrase table. It can be used in-memory and for on-demand loading. Like the original phrase table, this can only be used for phrase-based models. If you use this or the compact lexical reordering table below, please cite:

- Marcin Junczys-Dowmunt: **Phrasal Rank-Encoding: Exploiting Phrase Redundancy and Translational Relations for Phrase Table Compression**[24], Proceedings of the Machine Translation Marathon 2012, The Prague Bulletin of Mathematical Linguistics, vol. 98, pp. 63-74, 2012.

Download the CMPH library from http://sourceforge.net/projects/cmph/ and install according to the included instructions. Make sure the installation target directory contains an "include" and a "lib" directory. Next you need to recompile moses with

```
./bjam --with-cmph=/path/to/cmph
```

Now, you can convert the standard ascii phrase tables into the compact format. **Phrase tables are required to be sorted as above. For a maximal compression effect, it is advised to generate a phrase table with phrase-internal word alignment** (this is the default). If you want to compress a phrase table without alignment information, rather use `-encoding None` (see advanced options below). It is possible to use the default encoding (PREnc) without alignment information, but it will take much longer. For now, there may be problems to compact phrase tables on 32-bit systems since virtual memory usage quickly exceeds the 3 GB barrier.

Here is an example (standard phrase table phrase-table, with 5 scores) which produces a single file `phrase-table.minphr`:

```
mosesdecoder/bin/processPhraseTableMin -in phrase-table.gz -out phrase-table -nscores 5 -threads 4
```

In the moses config file, specify only the filename stem phrase-table as phrase table and set the type to 12, i.e.:

```
[ttable-file]
12 0 0 5 phrase-table
```

Options:

[24] http://ufal.mff.cuni.cz/pbml/98/art-junczys-dowmunt.pdf

```
-in  string       -- input table file name
-out string       -- prefix of binary table file
-nscores int      -- number of score components in phrase table
-alignment-info   -- include alignment info in the binary phrase table
-threads int      -- number of threads used for conversion
```

As for the original phrase table, the option `-alignment-info` includes phrase internal alignment information in the phrase table explicitly, without this option alignment data is used for compression only. However, regardless whether this option has been used, the resulting files can be employed with moses with or without the `-use-alignment-info` option.

Since compression is quite processor-heavy, it is advised to use the `-threads` option to increase speed.

Advanced options: Default settings should be fine for most of your needs, but the size of the phrase table can be tuned to your specific needs with the help of the advanced options.

```
-encoding string  -- encoding type: PREnc REnc None (default PREnc)
-rankscore int    -- score index of P(t|s) (default 2)
-maxrank int      -- maximum rank for PREnc (default 100)
-landmark int     -- use landmark phrase every 2\^n source phrases (default 10)
-fingerprint int  -- number of bits used for source phrase fingerprints (default 16)
-join-scores      -- single set of Huffman codes for score components
-quantize int     -- maximum number of scores per score component
-no-warnings      -- suppress warnings about missing alignment data
```

Encoding methods: There are two encoding types that can be used on-top the standard compression methods, Phrasal Rank-Encoding (`PREnc`) and word-based Rank Encoding (`REnc`). PREnc (see Junczys-Dowmunt (MTM 2012)[25] for details) is used by default and requires a phrase table with phrase-internal alignment to reach its full potential. PREnc can also work without explicit alignment information, but encoding is slower and the resulting file will be bigger, but smaller than without PREnc. The tool will warn you about every line that misses alignment information if you use PREnc or REnc. These warnings can be suppressed with `-no-warnings`. If you use PREnc with non-standard scores you should specify which score type is used for sorting with `-rankscore int`. By default this is P(t|s) which in the standard phrase table is the third score (index 2).

Basically with PREnc around, there is no reason to use REnc unless you really want to. It requires the lexical translation table `lex.f2e` to be present in the same directory as the text version phrase table. If no alignment information is available it falls back to None (See Junczys-Dowmunt (EAMT 2012)[26] for details on REnc and None).

[25] http://ufal.mff.cuni.cz/pbml/98/art-junczys-dowmunt.pdf

[26] http://hltshare.fbk.eu/EAMT2012/html/Papers/57.pdf

None is the fasted encoding method, but produces the biggest files. Concerning translation speed, there is virtually no difference between the encoding methods when the phrase tables are later used with moses, but smaller files result in lesser memory-usage, especially if the phrase tables are loaded entirely in-memory.

Indexing options: The properties of the source phrase index can be modified with the -landmark and -fingerprint options, changing these options can affect file size and translation quality, so do it carefully. Junczys-Dowmunt (TSD 2012)[27] contains a discussion of these values and their effects.

Scores and quantization: You can reduce the file size even more by using score quantization. E.g. with -quantize 1000000, a phrase table is generated that uses at most one million different scores for each score type. Be careful, low values will affect translation quality. By default, each score type is encoded with an own set of Huffman codes, with the -join-scores option only one set is used. If this option is combined with -quantize N, the summed number of different scores for all scores types will not exceed N.

In-memory loading: You can start moses with the option -minphr-memory to load the compact phrase table directly into memory at start up. Without this option, on-demand loading is used by default.

3.3.6 Compact Lexical Reordering Table

The compact lexical reordering table produces files about 12 to 15 times smaller than the original moses binary implementation. As for the compact phrase table you need to install CMPH and link against it. **Reordering tables must be sorted in the same way as the phrase tables above.** The command below produces a single file reordering-table.minlexr.

```
mosesdecoder/bin/processLexicalTableMin -in reordering-table.gz -out reordering-table -threads 4
```

If you include the prefix in the moses config file, the compact reordering table will be recognized and loaded automatically. You can start moses with the option -minlexr-memory to load the compact lexical reordering table directly into memory at start up.

Options: See the compact phrase table above for a description of available common options.

3.3.7 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.

[27] http://www.staff.amu.edu.pl/~junczys/dokuwiki/lib/exe/fetch.php?cache=&media=wiki:mjd2012tsd1.pdf

- `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`')

3.3.8 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-best-list 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 -1.08417 \
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-best-list 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 **??**) for further details).

3.3.9 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 **??**) and "Binary Phrase Table" (page **??**)), Moses can report them in the output.

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

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 1=1 2=1
0 ||| i ask you  ||| ...feature_scores.... ||| -3.1068 ||| 0-1=0-2 ||| 0=0 1=1,2 ||| 0=0 1=1 2=1
0 ||| i ask this  ||| ...feature_scores.... ||| -3.48919 ||| 0-1=0-2 ||| 0=0 1=1 ||| 0=0 1=1 2=-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 (required for binarized ttables, see "Binary Phrase Table" (page **??**)).
- `-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 n-best lists.
- `-alignment-output-file outfilename` – to output word-to-word alignments into a separate file in a compact format (one line per sentence).

3.3.10 Minimum Bayes Risk Decoding

Minumum Bayes Risk (MBR) decoding was proposed by Shankar Kumar and Bill Byrne (HLT/NAACL 2004)[28]. 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
```

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 to consider (default 200)
- `-mbr-scale` SCALE – scaling factor used to adjust the translation scores (default 1.0)

3.3.11 Lattice MBR and Consensus Decoding

These are extensions to MBR which may run faster or give better results. For more details see Tromble et al (2008)[29], Kumar et al (2009)[30] and De Nero et al (2009)[31]. The ngram posteriors

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

[29] http://www.aclweb.org/anthology-new/D/D08/D08-1065.pdf

[30] http://www.aclweb.org/anthology-new/P/P09/P09-1019.pdf

[31] http://www.aclweb.org/anthology-new/P/P09/P09-1064.pdf

(required for Lattice MBR) and the ngram expectations (for Consensus decoding) are both calculated using an algorithm descrived in De Nero et al (2010)[32]. Currently both lattice MBR and consensus decoding are implemented as n-best list rerankers, in other words the hypothesis space is an n-best list (not a lattice).

Here's the list of options which affect both lattice mbr and consensus-decoding.

- `-lmbr` – use Lattice MBR decoding
- `-con` – use Consensus decoding
- `-mbr-size` SIZE – as for MBR
- `-mbr-scale` SCALE – as for MBR
- `-lmbr-pruning-factor` FACTOR – mean words per node in pruned lattice, as described in Tromble et al (2008) (default 30)

Lattice MBR has several further parameters which are described in the Tromble et al 2008 paper.

- `-lmbr-p` P – The unigram precision (default 0.8)
- `-lmbr-r` R – The ngram precision ratio (default 0.6)
- `-lmbr-thetas` THETAS Instead of specifying p and r, lattice MBR can be configured by specifying all the ngram weights and the length penalty (5 numbers). This is described fully in the references.
- `-lmbr-map-weight` WEIGHT The weight given to the map hypothesis (default 0)

Since Lattice MBR has so many parameters, a utility to perform a grid search is provided. This is in `moses-cmd/src` and is called `lmbrgrid`. A typical usage would be

```
% ./lmbrgrid -lmbr-p 0.4,0.6,0.8 -lmbr-r 0.4,0.6,0.8 -mbr-scale 0.1,0.2,0.5,1 -lmbr-pruning-factor   \
30 -mbr-size 1000 -f moses.ini -i input.txt
```

In other words, the same Lattice MBR parameters as for `moses` are used, but this time a comma seperated list can be supplied. Each line in the output takes the following format:

```
<sentence-id> |||
 <r> <pruning-factor> <scale> ||| <translation>
```

In the moses Lattice MBR experiments that have been done to date, lattice MBR showed small overall improvements on a NIST Arabic data set (+0.4 over map, +0.1 over mbr), once the parameters were chosen carefully. Parameters were optimised by grid search on 500 sentences of heldout, and the following were found to be optimal

```
-lmbr-p 0.8 -lmbr-r 0.8 -mbr-scale 5 -lmbr-pruning-factor 50
```

[32] http://www.eecs.berkeley.edu/~denero/research/papers/naacl10_denero_combination.pdf

3.3.12 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

3.3.13 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
```

When using the switch -output-search-graph-extended (or short: -osgx), a detailed score breakdown is provided for each line. The format is the same as in the n-best list.

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
- `-output-search-graph-extended FILE` – output the search graph for each sentence in a file, with detailed feature breakdown

3.3.14 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)

3.3.15 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)

3.3.16 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

3.3.17 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 `,.!?:;"`.

3.3.18 Multiple Translation Tables and Backoff Models

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.

Backoff Models: You may want to prefer to use the first table, and the second table only if there are no translations to be found in the first table. In other words, the second table is only a backoff table for unknown words and phrases in the first table. This can be specified by the option decoding-graph-backoff. The option also allows if the backoff table should only be used for single words (unigrams), unigrams and bigrams, everything up to trigrams, up to 4-grams, etc.

For example, if you have two translation tables, and you want to use the second one only for unknown words, you would specify:

```
[decoding-graph-backoff]
0
1
```

The 0 indicates that the first table is used for anything (which it always should be), and the 1 indicates that the second table is used for unknown n-grams up to size 1. Replacing it with a 2 would indicate its use for unknown unigrams and bigrams (unknown in the sense that the first table has no translations for it).

Also note, that this option works also with more complicated mappings than just a single translation table. For instance the following specifies the use of a simple translation table first, and as a backoff a more complex factored decomposition involving two translation tables and two generation tables:

```
[mapping]
0 T 0
1 T 1
1 G 0
1 T 2
1 G 1

[decoding-graph-backoff]
0
1
```

3.3.19 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 `contrib/sigtest-filter`. You first need to build it from the source files.

This implementation relies on Joy Zhang's SALM Suffix Array toolkit[33].

[33] http://projectile.sv.cmu.edu/research/public/tools/salm/salm.htm#update

1. download and extract the SALM source release.
2. in `SALM/Distribution/Linux` type: `make`
3. enter the directory `contrib/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`).

```
% 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).

If you filter an hierarchical model, add the switch `-h`.

3.3.20 Pruning the Phrase Table based on Relative Entropy

While the pruning method in Johnson et al. 2007 is designed to remove spurious phrase pairs due to noisy data, it is also possible to remove phrase pairs that are redundant. That is, phrase pairs that can be composed by smaller phrase pairs in the model with similar probabilities. For more detail please refer to the following papers:

Ling, W., Graça, J., Trancoso, I., and Black, A. (2012). Entropy-based Pruning for Phrase-based Machine Translation. In Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), pp. 962-971.

Zens, R., Stanton, D., Xu, P. (2012). A Systematic Comparison of Phrase Table Pruning Technique. In Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), pp. 972-983.

The code from Ling et al. (2012)'s paper is available at `contrib/relent-filter`.

Build Instructions

The binaries for Relative Entropy-based Pruning are built automatically with moses. However, this implementation also calculates the significance scores (Johnson et al. 2007), using a slightly modified version of the code by Chris Dyer, which is in `contrib/relent-filter/sigtest-filter`. This must be built using the same procedure:

1. Download and build SALM available here[34]
2. Run "make `SALMDIR=/path/to/SALM`" in "`contrib/relent-filter/sigtest-filter`" to create the executable filter-pt

Usage Instructions

Checklist of required files (I will use <varname> to refer to these vars):

1. s_train - source training file
2. t_train - target training file
3. moses_ini - path to the moses configuration file (after tuning)
4. pruning_binaries - path to the relent pruning binaries (should be "`bin`" if no changes were made)
5. pruning_scripts - path to the relent pruning scripts (should be "`contrib/relent-filter/scripts`" if no changes were made)
6. sigbin - path to the sigtest filter binaries (should be "`contrib/relent-filter/sigtest-filter`" if no changes were made)
7. output_dir - path to write the output

Build suffix arrays for the source and target parallel training data

```
% SALM/Bin/Linux/Index/IndexSA.O32 <s_train>
% SALM/Bin/Linux/Index/IndexSA.O32 <t_train>
```

Calculate phrase pair scores by running:

```
% perl <pruning_scripts>/calcPruningScores.pl -moses_ini <moses_ini> -training_s <s_train> -training_t <t_train>
-prune_bin <pruning_binaries> -prune_scripts <pruning_scripts> -moses_scripts <path_to_moses>/scripts/training/ -workdir <output_dir> -dec_size 10000
```

This will create the following files in the <output_dir>/scores/ dir:

1. count.txt - counts of the phrase pairs for N(s,t) N(s,*) and N(*,t)
2. divergence.txt - negative log of the divergence of the phrase pair
3. empirical.txt - empirical distribution of the phrase pairs N(s,t)/N(*,*)
4. rel_ent.txt - relative entropy of the phrase pairs
5. significance.txt - significance of the phrase pairs

You can use any one of these files for pruning and also combine these scores using <pruning_scripts>/interpolate

To actually prune a phrase table, run <pruning_scripts>/prunePT.pl, this will prune phrase pairs based on the score file that is used. This script will prune the phrase pairs with lower scores first.

For instance, to prune 30% of the phrase table using relative entropy run:

[34] http://projectile.sv.cmu.edu/research/public/tools/salm/salm.htm#update

```
% perl <pruning_scripts>/prunePT.pl -table <phrase_table_file> -scores <output_dir>/scores/rel_ent.txt -percentage 70 > <pruned_phrase_table_file>
```

You can also prune by threshold

```
% perl <pruning_scripts>/prunePT.pl -table <phrase_table_file> -scores <output_dir>/scores/rel_ent.txt -threshold 0.1 > <pruned_phrase_table_file>
```

The same must be done for the reordering table by replacing <phrase_table_file> with the <reord_table_file>

```
% perl <pruning_scripts>/prunePT.pl -table <reord_table_file> -scores <output_dir>/scores/rel_ent.txt -percentage 70 > <pruned_reord_table_file>
```

Parallelization

The script <pruning_scripts>/calcPruningScores.pl requires the forced decoding of the whole set of phrase pairs in the phrase table, so unless it is used for a small corpora, it usually requires large amounts of time to process. Thus, we recommend users to run multiple instances of "<pruning_scripts>/calcPruningScores.pl" in parallel to process different parts of the phrase table.

To do this, run:

```
% perl <pruning_scripts>/calcPruningScores.pl -moses_ini <moses_ini> -training_s <s_train> -training_t <t_train>
-prune_bin <pruning_binaries> -prune_scripts <pruning_scripts> -moses_scripts <path_to_moses>/scripts/training/
-workdir <output_dir> -dec_size 10000 -start 0 -end 100000
```

The -start and -end options tell the script to only calculate the results for phrase pairs between 0 and 99999.

Thus, an example of a shell script to run for the whole phrase table would be:

```
size=‘wc <phrase_table_file> | gawk ’{print $1}’‘
phrases_per_process=100000

for i in $(seq 0 $phrases_per_process $size)
do
end=‘expr $i + $phrases_per_process‘
perl <pruning_scripts>/calcPruningScores.pl -moses_ini <moses_ini> -training_s <s_train> -training_t <t_train>
-prune_bin <pruning_binaries> -prune_scripts <pruning_scripts> -moses_scripts <path_to_moses>/scripts/training/
-workdir <output_dir>.$i-$end -dec_size 10000 -start $i -end $end
done
```

After all processes finish, simply join the partial score files together in the same order.

3.3.21 Multi-threaded Moses

Moses 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

Multi-threaded Moses is now built by default. If you omit the `-threads` argument, then Moses will use a single worker thread, and a thread to read the input stream. Using the argument `-threads n` specifies a pool of `n` threads, and `-threads all` will use all the cores on the machine.

3.3.22 Moses Server

The moses server enables you to run the decoder as a server process, and send it sentences to be translated via xmlrpc[35]. 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[36] 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 multi-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`).

Note that although the moses server needs to be built against multi-threaded moses, it can be run in single-threaded mode using the `-serial` option. This enables it to be used with non-threadsafe libraries such as (currently) irstlm.

Using Multiple Translation Systems in the Same Server

The moses server is now able to load multiple translation systems within the same server, and the client is able to decide which translation system that the server should use, on a per-sentence basis. The client does this by passing a `system` argument in the `translate` operation.

[35] http://www.xmlrpc.com/

[36] http://xmlrpc-c.sourceforge.net/

One possible use-case for this multiple models feature is if you want to build a server that translates both French and German into English, and uses a large English language model. Instead of running two copies of the moses server, each with a copy of the English language model in memory, you can now run one moses server instance, with the language model in memory, thus saving on RAM.

To use the multiple models feature, you need to make some changes to the standard moses configuration file. A sample configuration file can be found here[37].

The first piece of extra configuration required for a multiple models setup is to specify the available systems, for example

```
[translation-systems]
de D 0 R 0 L 0
fr D 1 R 1 L 1
```

This specifies that there are two systems (`de` and `fr`), and that the first uses decode path 0, reordering model 0, and language model 0, whilst the second uses the models with id 1. The multiple decode paths are specified with a stanza like

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

which indicates that the 0th decode path uses the 0th translation model, and the 1st decode path uses the 1st translation model. Using a language model specification like

```
[lmodel-file]
0 0 5 /disk4/translation-server/models/interpolated-lm
0 0 5 /disk4/translation-server/models/interpolated-lm
```

means that the same language model can be used in two different systems with two different weights, but moses will only load it once. The weights sections of the configuration file must have the correct numbers of weights for each of the models, and there must be a word penalty and linear distortion weight for each translation system. The lexicalised reordering weights (if any) must be specified in the `[weight-lr]` stanza, with the distortion penalty in the `[weight-d]` stanza.

[37] http://www.statmt.org/moses/img/moses-en.ini

3.3.23 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 SGE cluster. More details from the Amazon webpage, or from Achim directly[38]. Achim has also created a tutorial[39].

3.3.24 k-best batch MIRA Tuning

This is hope-fear MIRA built as a drop-in replacement for MERT; it conducts online training using aggregated k-best lists as an approximation to the decoder's true search space. This allows it to handle large features, and it often out-performs MERT once feature counts get above 10. The new code has been pushed into the master branch, as well as into miramerge.

You can tune using this system by adding `-batch-mira` to your mert-moses.pl command. This replaces the normal call to the mert executable with a call to kbmira.

I recommend also adding the flag `-return-best-dev` to mert-moses.pl. This will copy the moses.ini file corresponding to the highest-scoring development run (as determined by the evaluator executable using BLEU on run*.out) into the final moses.ini. This can make a fairly big difference for MIRA's test-time accuracy.

You can also pass through options to kbmira by adding `-batch-mira-args 'whatever'` to mert-moses.pl. Useful kbmira options include:

- `-J n` : changes the number of inner MIRA loops to n passes over the data. Increasing this value to 100 or 300 can be good for working with small development sets. The default, 60, is ideal for development sets with more than 1000 sentences.
- `-C n` : changes MIRA's C-value to n. This controls regularization. The default, 0.01, works well for most situations, but if it looks like MIRA is over-fitting or not converging, decreasing C to 0.001 or 0.0001 can sometimes help.
- `-streaming` : stream k-best lists from disk rather than load them into memory. This results in very slow training, but may be necessary in low-memory environments or with very large development sets.

Run kbmira `-help` for a full list of options.

So, a complete call might look like this:

$MOSES_SCRIPTS/training/mert-moses.pl work/dev.fr work/dev.en $MOSES_BIN/moses work/model/mose
–mertdir $MOSES_BIN –rootdir $MOSES_SCRIPTS –batch-mira –return-best-dev –batch-mira-args '-J 300' –decoder-flags '-threads 8 -v 0'

Please give it a try. If it's not working as advertised, send Colin Cherry an e-mail.

For more information on batch MIRA, check out our paper:

Colin Cherry and George Foster Batch Tuning Strategies for Statistical Machine Translation NAACL, June 2012

https://sites.google.com/site/colinacherry/Cherry_Foster_NAACL_2012.pdf

Anticipating some questions:

[38] http://developer.amazonwebservices.com/connect/entry.jspa?externalID=3058&ca

[39] http://www.digitalsilkroad.net/walkthrough.pdf

[Q: Does it only handle BLEU?] [A: Yes, for now. There's nothing stopping people from implementing other metrics, so long as a reasonable sentence-level version of the metric can be worked out. Note that you generally need to retune kbmira's C-value for different metrics. I'd also change –return-best-dev to use the new metric as well.]

[Q: Have you tested this on a cluster?] [A: No, I don't have access to a Sun Grid cluster - I would love it if someone would test that scenario for me. But it works just fine using multi-threaded decoding. Since training happens in a batch, decoding is embarrassingly parallel.]

3.3.25 ZMERT Tuning

Kamil Kos has created `scripts/training/zmert-moses.pl`, a replacement for `mert-moses.pl` for those who wish to use ZMERT. The `zmert-moses.pl` script supports most of the `mert-moses.pl` parameters, therefore the transition to the new zmert version should be relatively easy. For more details on supported parameters run `scripts/training/zmert-moses.pl -help`.

The main advantage of ZMERT is that, in common with new mert, it can optimize towards different metrics (BLEU and TER are built-in). How to implement new metrics is described on ZMERT homepage[40]. At Charles University in Prague, we're experimenting with SemPOS[41] which is based on the tectogrammatical layer, see TectoMT[42].

ZMERT JAR, version 1.41 needs to be downloaded from Omar Zaidan's website[43]. If you would like to add a new metric, please, modify the `zmert/zmert.jar` file in the following way:

1. extract `zmert.jar` content by typing `jar xf zmert.jar`
2. modify the files (probably a copy of `NewMetric.java.template`)
3. recompile java files by `javac *.java`
4. create a new version of `zmert.jar` by typing `jar cvfM zmert.jar *.java* *.class`

3.3.26 Lattice MERT

This was implemented by Kārlis Goba and Christian Buck at the Fourth Machine Translation Marathon in January 2010. It is based on the work of Macherey et al[44] and is available here[45].

3.3.27 Continue Partial Translation

This option forces Moses to start generating the translation from a non-empty hypothesis. This can be useful in situations, when you have already translated some part of the sentence and want to get a suggestion or an n-best-list of continuations.

Use `-continue-partial-translation` (`-cpt`) to activate this feature. With `-cpt`, moses accepts also a special format of the input: three parameters delimited by the triple bar (|||). The first

[40] http://www.cs.jhu.edu/~ozaidan/zmert/
[41] http://ufal.mff.cuni.cz/pbml/92/art-pbml92-kos-bojar.pdf
[42] http://ufal.mff.cuni.cz/tectomt
[43] http://www.cs.jhu.edu/~ozaidan/zmert/
[44] http://research.google.com/pubs/pub34629.html
[45] https://github.com/christianbuck/Moses-Lattice-MERT

parameter is the string of output produced so far (used for LM scoring). The second parameter is the coverage vector of input words are already translated by the output so far, written as a string of "1"s and "0"s of the same length as there are words in the input sentence. The third parameter is the source sentence.

Example:

```
% echo "that is ||| 11000 ||| das ist ein kleines haus" | moses -f moses.ini -continue-partial-translation
that is a small house

% echo "that house ||| 10001 ||| das ist ein kleines haus" | moses -f moses.ini -continue-partial-translation
that house is a little
```

If the input does not fit to this pattern, it is treated like normal input with no words translated yet.

This type of input is currently **not** compatible with factored models or confusion networks. The standard non-lexicalized distortion works but more or less as one would expect (note that some input coverage vectors may prohibit translation under low distortion limits). The lexicalized reordering has not been tested.

Options

- `-continue-partial-translation` (`-cpt`) – activate the feature

3.3.28 Global Lexicon Model

The global lexicon model predicts the bag of output words from the bag of input words. It does not use an explicit alignment between input and output words, so word choice is also influenced by the input context. For details, please check Arne Mauser's paper[46].

The model is trained with the script

```
scripts/training/train-global-lexicon-model.perl --corpus-stem FILESTEM --lex-dir DIR --f EXT --e EXT
```

which requires the tokenized parallel corpus, and the lexicon files required for GIZA++.

You will need the MegaM[47] maximum entropy classifier from Hal Daume for training.

Warning: A separate maximum entropy classifier is trained for each target word, which is very time consuming. The training code is a very experimental state. It is very inefficient. For instance training a model on Europarl German-English with 86,700 distinct English words took about 10,000 CPU hours.

The model is stored in a text file.

File format:

[46] http://www-i6.informatik.rwth-aachen.de/publications/download/628/Mauser-EMNLP-2009.pdf

[47] http://www.cs.utah.edu/~hal/megam/

```
county initiativen 0.34478
county land 0.92405
county schaffen 0.23749
county stehen 0.39572
county weiteren 0.04581
county europa -0.47688
```

Specification in `moses.ini`:

```
[global-lexical-file]
0-0 /home/pkoehn/experiment/discr-lex/model/global-lexicon.gz

[weight-lex]
0.1
```

3.3.29 Incremental Training

Introduction

Translation models for Moses are typically batch trained. That is, before training you have all the data you wish to use, you compute the alignments using GIZA, and from that produce a phrase table which you can use in the decoder. If some time later you wish to utilize some new training data, you must repeat the process from the start, and for large data sets, that can take quite some time.

Incremental training provides a way of avoiding having to retrain the model from scratch every time you wish to use some new training data. Instead of producing a phrase table with precalculated scores for all translations, the entire source and target corpora are stored in memory as a suffix array along with their alignments, and translation scores are calculated on the fly. Now, when you have new data, you simply update the word alignments, and append the new sentences to the corpora along with their alignments. Moses provides a means of doing this via XML RPC, so you don't even need to restart the decoder to use the new data.

Note that at the moment the incremental phrase table code is not thread safe.

Initial Training

This section describes how to initially train and use a model which support incremental training.

- Setup the EMS as normal, but use the version of GIZA++ found at http://code.google.com/p/inc-giza-pp/
- Add the line:

```
train-model-options = "-final-alignment-model hmm"
```

to the TRAINING section of your experiment configuration file.

- Train the system using the initial training data as normal.
- Modify the moses.ini file found in *<experiment-dir>/evaluation/filtered.<evaluation-set>.<run-number>* to have a ttable-file entry as follows:

```
8 0 0 3 <path-to-src> <path-to-tgt> <path-to-alignments>
```

The source and target corpus paths should be to the tokenized, cleaned, and truecased versions found in *<experiment-dir>/training/corpus.<run>.<lang>*, and the alignment path should be to *<experiment-dir>/model/aligned.<run>.grow-diag-final-and*.

Updates

- Preprocess New Data

First, tokenise, clean, and truecase both target and source sentences (in that order) in the same manner as for the original corpus. You can see how this was done by looking at the *<experiment-dir>/steps/<run>/CORPUS_{tokenize,clean,truecase}.<run>* scripts.

- Prepare New Data

The preprocessed data now needs to be prepared for use by GIZA. This involves updating the vocab files for the corpus, converting the sentences into GIZA's *snt* format, and updating the cooccurrence file.

- plain2snt

```
$ plain2snt.out txt1 txt2 [txt3 txt4 -weight w -txt1-vocab v1 -txt2-vocab v2]
```

The vocab files for the original corpus can be found in *<experiment-dir>/training/prepared.<run>/<source-lang>,<target-lang>.vcb*. Running this command with the files containing your new tokenized, cleaned, and truecased source and target as *txt1* and *txt2* will produce new a new vocab file for each language and a couple of *.snt* files. Any further references to vocabs in commands or config files should reference the ones just produced.

- snt2cooc

```
$ ./snt2cooc.out vcb1 vcb2 snt12 [prevCoocFile]
```

This commands should be run twice; once in the source-target direction, and once in the target source direction. The previous cooccurrence files can be found in *<experiment-dir>/training/giza.<run>/<source-lang>-<target-lang>.cooc* and *<experiment-dir>/training/giza-inverse.<run>/<target-lang>-<source-lang>.cooc*.

- Update and Compute Alignments

GIZA++ can now be run to update and compute the alignments for the new data. This should be run in the source to target, and target to source directions. A sample GIZA++ config file is given below for the source to target direction; for the target to source direction, simply swap mentions of target and source.

```
S: <path-to-src-vocab>
T: <path-to-tgt-vocab>
C: <path-to-src-to-tgt-snt>
O: <prefix-of-output-files>
coocurrencefile: <path-to-src-tgt-cooc-file>
model1iterations: 1
model1dumpfrequency: 1
hmmiterations: 1
hmmdumpfrequency: 1
model2iterations: 0
model3iterations: 0
model4iterations: 0
model5iterations: 0
emAlignmentDependencies: 1
step_k: 1
oldTrPrbs: <path-to-original-thmm>
oldAlPrbs: <path-to-original-hhmm>
```

To run GIZA++ with these config files, just issue the command

```
GIZA++ <path-to-config-file>
```

With the alignments updated, we can get the alignments for the new data by running the command:

giza2bal.pl -d <path-to-updated-tgt-to-src-ahmm> -i <path-to-updated-src-to-tgt-ahmm> | symal -alignment="grow" -diagonal="yes" -final="yes" -both="yes" > new-alignment-file

- Update Model

Now that alignments have been computed for the new sentences, you can use them in the decoder. Updating a running Moses instance is done via XML RPC, however to make the changes permanent, you must append the tokenized, cleaned, and truecased source and target sentences to the original corpora, and the new alignments to the alignment file.

3.3.30 Distributed Language Model

In most cases, MT output improves significantly when more data is used to train the Language Model. More data however produces larger models, and it is very easy to produce a model which cannot be held in the main memory of a single machine. To overcome this, the Language Model can be distributed across many machines, allowing more data to be used at the cost of a performance overhead.

Support for Distributed Language Models in Moses are built on top of a bespoke distributed map implementation called DMap. DMap and support for Distributed Language Models are still in beta, and any feedback or bug reports are welcomed.

Installing and Compiling

Before compiling Moses with DMap support, you must configure your DMap setup (see below). Once that has been done, run Moses' *configure* script with your normal options and *–with-dmaplm=<path-to-dmap>*, then the usual *make*, *make install*.

Configuration

Configuring DMap is at the moment, a very crude process. One must edit the *src/DMap/Config.cpp* file by hand and recompile when making any changes. With the configuration being compiled in, this also means that once it is changed, any programs statically linked to DMap will have to be recompiled too. The file *src/DMap/Config.cpp* provides a good example configuration which is self explanatory.

Example

In this example scenario, we have a Language Model trained on the *giga4* corpus which we wish to host across 4 servers using DMap. The model is a 5-gram model, containing roughly 210 million ngrams; the probabilities and backoff weights of ngrams will be uniformly quantised to 5 bit values.

1. Configuration

Here is an example Config.cpp for such a set up:

```
config->setShardDirectory("/home/user/dmap");
config->addTableConfig(new TableConfigLossyDoubleHash(
"giga4",    // name of table
283845991,  // number of cells (approx 1.23 * number of ngrams)
64,         // number of chunks (not too important, leave at 64)
(((uint64_t)1 << 61) - 1),             // universal hashing P parameter
5789372245 % (((uint64_t)1 << 61) - 1), // universal hashing a parameter
3987420741 % (((uint64_t)1 << 61) - 1), // universal hashing b parameter
"/home/user/dmap/giga4.bf",
16,         // num_error_bits (higher -> fewer collisions but more memory)
10,         // num_value_bits (higher -> more accurate probabilities and backoff weights but more memory)
20));       // num_hashes (higher ->
config->addStructConfig(new StructConfigLanguageModelBackoff(
"giga4",    // struct name
"giga4",    // lm table name
5,          // lm order
5,          // num logprob bits (these fields should add up to the number of value bits for the table)
5));        // num backoff bits
config->addServerConfig(new ServerConfig("server0.some.domain", 5000));
config->addServerConfig(new ServerConfig("server1.some.domain", 5000));
config->addServerConfig(new ServerConfig("server2.some.domain", 5000));
config->addServerConfig(new ServerConfig("server3.some.domain", 5000));
```

Note that the shard directory should be on a shared file system all Servers can access.

1. Create Table

The command:

```
create_table giga4
```

will create the files for the shards.

1. Shard Model

The model can now be split into chunks using the *shard* utility:

```
shard giga4 /home/user/dmap/giga4.arpa
```

1. Create Bloom Filter

A Bloom Filter is a probabilist data structure encoding set membership in an extremely space efficient manner. When querying whether a given item is present in the set they enocde, they can produce an error with a calculable probability. This error is one sided in that they can produce false positives, but never false negatives. To avoid making slow network requests, DMap keeps a local bloom filter containing the set of ngrams in the Language Model. Before making a network request to get the probability of an ngram, DMap first checks to see if the ngram is present in the Bloom Filter. If is not, then we know for certain the ngram is not present in the model and therefore not worth issuing a network request for. However, if the ngram is present in the filter, it might actually be in the model, or the filter may have produced a false positive.

To create a Bloom Filter containing the ngrams of the Language Model, run this command:

```
ngrams < /home/user/dmap/giga4.arpa | mkbf 134217728 210000000 /home/user/dmap/giga4.bf
```

1. Integration with Moses

The structure within DMap Moses should use as the Language Model should be put into a file, in this case at /home/user/dmap/giga4.conf:

```
giga4
false
```

Note that if for testing or experimentation purposes you would like to have the whole model on the local machine instead of over the network, change the false to true. You must have sufficient memory to host the whole model, but decoding will be significantly faster.

To use this, put the following line in your moses.ini file:

```
11 0 0 5 /home/user/dmap/giga4.conf
```

3.3.31 Suffix Arrays for Hierarchical Models

The phrase-based model uses a suffix array implementation which comes with Moses.

If you want to use suffix arrays for hierarchical models, use Adam Lopez's implementation. The source code for this is currently in cdec at

```
https://github.com/redpony/cdec/
```

unde the subdirectory

```
sa-extract
```

You have to compile cdec so please follow it's instructions.

Note - the suffix array code requires Python 2.7 or above, and cython . If you have Linux installations which are a few years old, check this first.

Adam's implementation writes the suffix array to binary files, given the parallel training data and word alignment. The Moses toolkit has a wrapper script which simplifies this process:

```
./scripts/training/wrappers/adam-suffix-array/suffix-array-create.sh
[path to cdec/sa-extract]
[source corpus]
[target corpus]
[word alignment]
[output suffix array directory]
[output glue rules]
```

WARNING - This requires a lot of memory (approximately 10GB for a parallel corpus of 15 million sentence pairs)

Once the suffix array has been created, run another Moses wrapper script to extract the translation rules required for a particular set of input sentences.

```
./scripts/training/wrappers/adam-suffix-array/suffix-array-extract.sh
[suffix array directory from previous command]
[input sentences]
[output rules directory]
```

This command creates one file for each input sentences with just the rules required to decode that sentences. eg.

```
# ls filtered.5/
grammar.out.0.gz        grammar.out.3.gz        grammar.out.7.gz
grammar.out.1.gz        grammar.out.4.gz        grammar.out.8.gz
grammar.out.10.gz        grammar.out.5.gz        grammar.out.9.gz ....
```

Note - these files are gzipped, and the rules are formatted in the Hiero format, rather than the Moses format. eg.

```
# zcat filtered.5/grammar.out.0.gz | head -1
[X] ||| monsieur [X,1] ||| mr [X,1] ||| 0.178069829941 2.04532289505 1.8692317009 0.268405526876 0.160579100251 0.0 0.0 ||| 0-0
```

To use these rules in the decoder, put this into the ini file

```
[ttable-file]
10 0 0 7 [rules directory from previous command]
```

Using the EMS

Adam's suffix array implementation is integrated into the EMS, where all of the above commands are executed for you. Add the following line to your EMS config file:

```
[TRAINING]
suffix-array = [sa-extract path]
```

and the EMS will use the suffix array instead of the usual Moses rule extraction algorithms.

3.3.32 Fuzzy Match Rule Table for Hierachical Models

Another method of extracting rules from parallel data is described in (Koehn, Senellart, 2010-1 AMTA)[48] and (Koehn, Senellart, 2010-2 AMTA)[49].

To use this extraction method in the decoder, add this to the moses.ini file:

```
[ttable-file]
11 0 0 3 source.path;target.path;alignment
```

It has not yet been integrated into the EMS.

Note - the translation rules generated by this algorith is intended to be used in the chart decoder. It can't be used in the phrase-based decoder.

3.3.33 Translation Model Combination

You can combine several phrase tables by linear interpolation or instance weighting using the script ***contrib/tmcombine/tmcombine.py***, or by fill-up using the script ***contrib/combine-ptables/combine-ptables.pl***.

Linear Interpolation and Instance Weighting

Linear interpolation works with any models; for instance weighting, models need to be trained with the option `-write-lexical-counts` so that all sufficient statistics are available. You can set corpus weights by hand, and instance weighting with uniform weights corresponds to a concatenation of your training corpora (except for differences in word alignment).

You can also set weights automatically so that perplexity on a tuning set is minimized. To obtain a tuning set from a parallel tuning corpus, use the Moses training pipeline to automatically extract a list of phrase pairs. The file *model/extract.sorted.gz* is in the right format.

An example call: (this weights *test/model1* and *test/model2* with instance weighting (-m counts) and *test/extract* as development set for perplexity minimization, and writes the combined phrase table to *test/phrase-table_test5*)

```
python tmcombine.py combine_given_tuning_set test/model1 test/model2 -m counts -o test/phrase-table_test5 -r test/extract
```

More information is available in (Sennrich, 2012 EACL)[50] and *contrib/tmcombine/README.md*.

Fill-up

[48] http://homepages.inf.ed.ac.uk/pkoehn/publications/amta2010.pdf

[49] http://homepages.inf.ed.ac.uk/pkoehn/publications/tm-smt-amta2010.pdf

[50] http://www.aclweb.org/anthology/E/E12/E12-1055.pdf

This combination technique is useful when the relevance of the models is known a priori: typically, when one is trained on in-domain data and the others on out-of-domain data.

Fill-up preserves all the entries and scores coming from the first model, and adds entries from the other models only if new. Moreover, a binary feature is added for each additional table to denote the provenance of an entry. These binary features work as scaling factors that can be tuned directly by MERT along with other models' weights.

Fill-up can be applied to both translation and reordering tables.

Example call, where ptable0 is the in-domain model:

```
perl combine-ptables.pl --mode=fillup ptable0 ptable1 ... ptableN > ptable-fillup
```

More information is available in (Bisazza et al., 2011 IWSLT)[51] and *contrib/combine-ptables/README.md*.

Subsection last modified on November 21, 2012, at 04:35 PM

3.4 Experiment Management System

- Introduction (page **??**)
- Requirements (page **??**)
- Quick Start (page **??**)
- More Examples (page **??**)
- Phrase Model (page **??**)
- Factored Phrase Model (page **??**)
- Hierarchical model (page **??**)
- Target syntax model (page **??**)
- Try a Few More Things (page **??**)
- Stemmed Word Alignment (page **??**)
- Using Multi-Threaded GIZA++ (page **??**)
- Using Berkeley Aligner (page **??**)
- IRST Language Model (page **??**)
- Randomized Language Model (page **??**)
- Compound Splitting (page **??**)
- A Short Manual (page **??**)
- Experiment.Meta (page **??**)

[51] http://www.mt-archive.info/IWSLT-2011-Bisazza.pdf

- Multiple Corpora, One Translation Model (page **??**)

- Defining Settings (page **??**)

- Working with Experiment.Perl (page **??**)

- Continuing Crashed Experiments (page **??**)

- Running on a Cluster (page **??**)

- Running on a Multi-core Machine (page **??**)

- Web Interface (page **??**)

- Analysis (page **??**)

- Segmentation (page **??**)

- Coverage Analysis (page **??**)

- Bilingual Concordancer (page **??**)

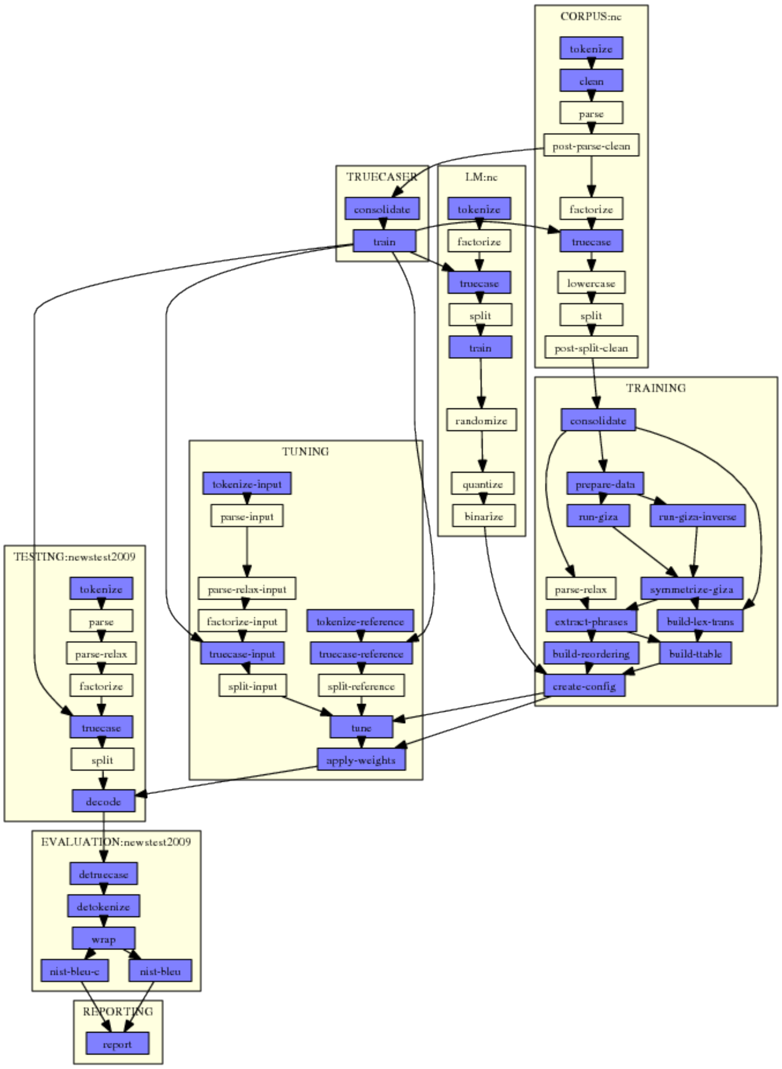
- Precision by coverage (page **??**)

3.4.1 Introduction

The Experiment Management System (EMS), or Experiment.perl, for lack of a better name, makes it much easier to perform experiments with Moses.

There are many steps in running an experiment: the preparation of training data, building language and translation models, tuning, testing, scoring and analysis of the results. For most of these steps, a different tool needs to be invoked, so this easily becomes very messy.

Here a typical example:

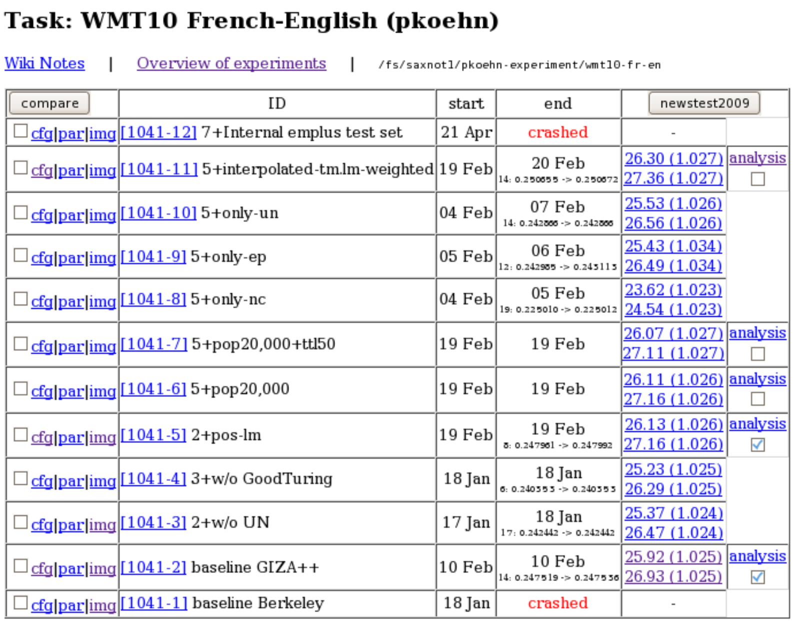


This graph was automatically generated by Experiment.perl. All that needed to be done was to specify one single configuration file that points to data files and settings for the experiment.

In the graph, each step is a small box. For each step, Experiment.perl builds a script file that gets either submitted to the cluster or run on the same machine. Note that some steps are quite involved, for instance tuning: On a cluster, the tuning script runs on the head node a submits jobs to the queue itself.

Experiment.perl makes it easy to run multiple experimental runs with different settings or data resources. It automatically detects which steps do not have to be executed again but instead which results from an earlier run can be re-used.

Experiment.perl also offers a web interface to the experimental runs for easy access and comparison of experimental results.

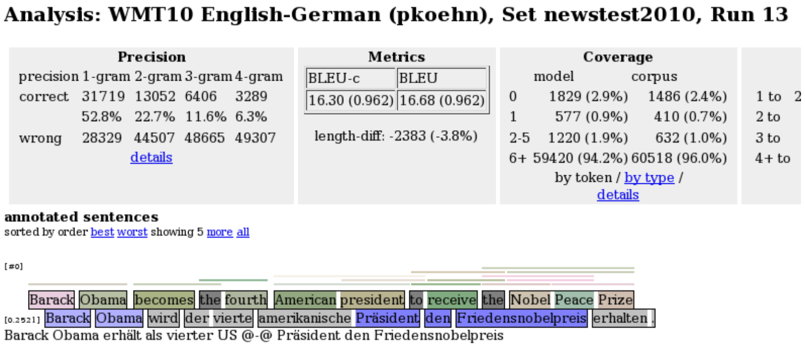


Task: WMT10 French-English (pkoehn)

Wiki Notes | Overview of experiments | /fs/saxnot1/pkoehn-experiment/wmt10-fr-en

compare	ID	start	end	newstest2009	
□ cfg|par|img	[1041-12] 7+Internal emplus test set	21 Apr	crashed	-	
□ cfg|par|img	[1041-11] 5+interpolated-tm.lm-weighted	19 Feb	20 Feb 14: 0.250655 -> 0.250672	26.30 (1.027) 27.36 (1.027)	analysis □
□ cfg|par|img	[1041-10] 5+only-un	04 Feb	07 Feb 14: 0.242866 -> 0.242866	25.53 (1.026) 26.56 (1.026)	
□ cfg|par|img	[1041-9] 5+only-ep	05 Feb	06 Feb 12: 0.242985 -> 0.243113	25.43 (1.034) 26.49 (1.034)	
□ cfg|par|img	[1041-8] 5+only-nc	04 Feb	05 Feb 19: 0.225010 -> 0.225012	23.62 (1.023) 24.54 (1.023)	
□ cfg|par|img	[1041-7] 5+pop20,000+ttl50	19 Feb	19 Feb	26.07 (1.027) 27.11 (1.027)	analysis □
□ cfg|par|img	[1041-6] 5+pop20,000	19 Feb	19 Feb	26.11 (1.026) 27.16 (1.026)	analysis □
□ cfg|par|img	[1041-5] 2+pos-lm	19 Feb	19 Feb 8: 0.247961 -> 0.247992	26.13 (1.026) 27.16 (1.026)	analysis ☑
□ cfg|par|img	[1041-4] 3+w/o GoodTuring	18 Jan	18 Jan 6: 0.240353 -> 0.240353	25.23 (1.025) 26.29 (1.025)	
□ cfg|par|img	[1041-3] 2+w/o UN	17 Jan	18 Jan 17: 0.242442 -> 0.242442	25.37 (1.024) 26.47 (1.024)	
□ cfg|par|img	[1041-2] baseline GIZA++	10 Feb	10 Feb 14: 0.247519 -> 0.247536	25.92 (1.025) 26.93 (1.025)	analysis ☑
□ cfg|par|img	[1041-1] baseline Berkeley	18 Jan	crashed	-	

The web interface also offers some basic analysis of results, such as comparing the n-gram matches between two different experimental runs:



Analysis: WMT10 English-German (pkoehn), Set newstest2010, Run 13

Precision

precision	1-gram	2-gram	3-gram	4-gram
correct	31719	13052	6406	3289
	52.8%	22.7%	11.6%	6.3%
wrong	28329	44507	48665	49307

details

Metrics

BLEU-c	BLEU
16.30 (0.962)	16.68 (0.962)

length-diff: -2383 (-3.8%)

Coverage

	model	corpus	
0	1829 (2.9%)	1486 (2.4%)	1 to 2
1	577 (0.9%)	410 (0.7%)	2 to
2-5	1220 (1.9%)	632 (1.0%)	3 to
6+	59420 (94.2%)	60518 (96.0%)	4+ to

by token / by type /
details

annotated sentences
sorted by order best worst showing 5 more all

[#0]
Barack Obama becomes the fourth American president to receive the Nobel Peace Prize
[0.2521] Barack Obama wird der vierte amerikanische Präsident den Friedensnobelpreis erhalten .
Barack Obama erhält als vierter US @-@ Präsident den Friedensnobelpreis

3.4.2 Requirements

In order to run properly, EMS will require:

- A version of Moses along with SRILM,
- The GraphViz toolkit[52],
- The ImageMagick toolkit[53], and
- The GhostView tool[54].

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

[53] http://www.imagemagick.org/script/index.php

[54] http://www.gnu.org/software/gv/

3.4.3 Quick Start

Experiment.perl is extremely simple to use:

- Find `experiment.perl` in `scripts/ems`
- Get a sample configuration file from someplace (for instance `scripts/ems/example/config.toy`).
- Set up a working directory for your experiments for this task (`mkdir` does it).
- Edit the following path settings in `config.toy`
  - `working-dir`
  - `data-dir`
  - `moses-script-dir`
  - `moses-src-dir`
  - `srilm-dir`
  - `decoder`
- Run `experiment.perl -config config.toy` from your experiment working directory.
- Marvel at the graphical plan of action.
- Run `experiment.perl -config config.toy -exec`.
- Check the results of your experiment (in `evaluation/report.1`)

Let us take a closer look at what just happened.

The configuration file `config.toy` consists of several sections. For instance there is a section for each language model corpus to be used. In our toy example, this section contains the following:

```
[LM:toy]

### raw corpus (untokenized)
#
raw-corpus = $toy-data/nc-5k.$output-extension
```

The setting `raw-corpus` species the location of the corpus. The definition uses the variables `$toy-data` and `$output-extension`, which are also settings defined elsewhere in the configuration file. These variables are resolved, leading to the file path `ems/examples/data/nc-5k.en` in your Moses scripts directory.

The authoritative definition of the steps and their interaction is in the file `experiment.meta` (in the same directory as experiment.perl: `scripts/ems`).

The logic of `experiment.meta` is that it wants to create a report at the end. To generate the report it needs to evaluation scores, to get these it needs decoding output, to get these it needs to run the decoder, to be able to run the decoder it needs a trained model, to train a model it needs data. This process of defining the agenda of steps to be executed is very similar to the Make utility in Unix.

We can find the following step definitions for the language model module in `experiment.meta`:

```
get-corpus
in: get-corpus-script
out: raw-corpus
default-name: lm/txt
template: IN > OUT
tokenize
in: raw-corpus
out: tokenized-corpus
default-name: lm/tok
pass-unless: output-tokenizer
template: $output-tokenizer < IN > OUT
parallelizable: yes
```

The tokenization step `tokenize` requires `raw-corpus` as input. In our case, we specified the setting in the configuration file. We could have also specified an already tokenized corpus with `tokenized-corpus`. This would allow us to skip the tokenization step. Or, to give another example, we could have not specified `raw-corpus`, but rather specify a script that generates the corpus with the setting `get-corpus-script`. This would have triggered the creation of the step `get-corpus`.

The steps are linked with the definition of their input `in` and output `out`. Each step has also a default name for the output (`efault-name`) and other settings.

The tokenization step has as default name `lm/tok`. Let us look at the directory `lm` to see which files it contains:

```
% ls -tr lm/*
lm/toy.tok.1
lm/toy.truecased.1
lm/toy.lm.1
```

We find the output of the tokenization step in the file `lm/toy.tok.1`. The `toy` was added from the name definition of the language model (see `[LM:toy]` in `config.toy`). The `1` was added, because this is the first experimental run.

The directory `steps` contains the script that executes each step, its STDERR and STDOUT output, and meta-information. For instance:

```
% ls steps/1/LM_toy_tokenize.1* | cat
steps/1/LM_toy_tokenize.1
steps/1/LM_toy_tokenize.1.DONE
steps/1/LM_toy_tokenize.1.INFO
steps/1/LM_toy_tokenize.1.STDERR
steps/1/LM_toy_tokenize.1.STDERR.digest
steps/1/LM_toy_tokenize.1.STDOUT
```

The file `steps/2/LM_toy_tokenize.2` is the script that is run to execute the step. The file with the extension `DONE` is created when the step is finished - this communicates to the scheduler that subsequent steps can be executed. The file with the extension `INFO` contains meta-information - essential the settings and dependencies of the step. This file is checked to detect if a step can be re-used in new experimental runs.

In case that the step crashed, we expect some indication of a fault in `STDERR` (for instance the words `core dumped` or `killed`). This file is checked to see if the step was executed successfully, so subsequent steps can be scheduled or the step can be re-used in new experiments. Since the `STDERR` file may be very large (some steps create megabytes of such output), a digested version is created in `STDERR.digest`. If the step was successful, it is empty. Otherwise it contains the error pattern that triggered the failure detection.

Let us now take a closer look at re-use. If we run the experiment again but change some of the settings, say, the order of the language model, then there is no need to re-run the tokenization.

Here is the definition of the language model training step in `experiment.meta`:

```
train
in: split-corpus
out: lm
default-name: lm/lm
ignore-if: rlm-training
rerun-on-change: lm-training order settings
template: $lm-training -order $order $settings -text IN -lm OUT
error: cannot execute binary file
```

The mention of `order` in the list behind `rerun-on-change` informs experiment.perl that this step does need to be re-run, of the order of the language model changes. Since none of the settings in the chain of steps leading up to the training have been changed, the can be re-used.

Try changing the language model order (`order = 5` in `config.toy`), run experiment.perl again (`experiment.perl -config config.toy`) in the working directory, and you will see the new language model in the directory `lm`:

```
% ls -tr lm/*
lm/toy.tok.1
lm/toy.truecased.1
lm/toy.lm.1
lm/toy.lm.2
```

3.4.4 More Examples

The `example` directory contains some additional examples.

These require the training and tuning data released for the Shared Translation Task for WMT 2010. Create a working directory, and change into it. Then execute the following steps:

```
mkdir data
cd data
wget http://www.statmt.org/wmt10/training-parallel.tgz
tar xzf training-parallel.tgz
wget http://www.statmt.org/wmt10/dev.tgz
tar xzf dev.tgz
cd ..
```

The examples using these corpora are

- `config.basic` - a basic phrase based model,
- `config.factored` - a factored phrase based model,
- `config.hierarchical` - a hierarchical phrase based model, and
- `config.syntax` - a target syntax model.

In all these example configuration files, most corpora are commented out. This is done by adding the word `IGNORE` at the end of a corpus definition (also for the language models). This allows you to run a basic experiment with just the News Commentary corpus which finished relatively quickly. Remove the `IGNORE` to include more training data. You may run into memory and disk space problems when using some of the larger corpora (especially the news language model), depending on your computing infrastructure.

If you decide to use multiple corpora for the language model, you may also want to try out interpolating the individual language models (instead of using them as separate feature functions). For this, you need to comment out the `IGNORE` next to the `[INTERPOLATED-LM]` section.

You may also specify different language pairs by changing the `input-extension`, `output-extension`, and `pair-extension` settings.

Finally, you can run all the experiments with the different given configuration files and the data variations in the same working directory. The experimental management system figures out automatically which processing steps do not need to repeated because they can be re-used from prior experimental runs.

Phrase Model

Phrase models are, compared to the following examples, the simplest models to be trained with Moses and the fastest models to run. You may prefer these models over the more sophisticated models whose added complexity may not justify the small (if any) gains.

The example `config.basic` is similar to the toy example, except for a larger training and test corpora. Also, the tuning stage is not skipped. Thus, even with most of the corpora commented out, the entire experimental run will likely take a day, with most time taken up by word alignment (`TRAINING_run-giza` and `TRAINING_run-giza-inverse`) and tuning (`TUNING_tune`).

Factored Phrase Model

Factored models allow for additional annotation at the word level which may be exploited in various models. The example in `config.factored` uses part-of-speech tags on the English target side.

Annotation with part-of-speech tags is done with MXPOST, which needs to be installed first. Please read the installation instructions[55]. After this, you can run experiment.perl with the configuration file `config.factored`.

If you compare the factored example `config.factored` with the phrase-based example `config.basic`, you will notice the definition of the factors used:

```
### factored training: specify here which factors used
# if none specified, single factor training is assumed
# (one translation step, surface to surface)
#
input-factors = word
output-factors = word pos
alignment-factors = "word -> word"
translation-factors = "word -> word+pos"
reordering-factors = "word -> word"
#generation-factors =
decoding-steps = "t0"
```

the factor definition:

```
#################################################################
# FACTOR DEFINITION

[INPUT-FACTOR]

# also used for output factors
temp-dir = $working-dir/training/factor

[OUTPUT-FACTOR:pos]

### script that generates this factor
#
mxpost = /home/pkoehn/bin/mxpost
factor-script = "$moses-script-dir/training/wrappers/make-factor-en-pos.mxpost.perl -mxpost $mxpost"
```

and the specification of a 7-gram language model over part of speech tags:

[55] http://www.statmt.org/moses/?n=Moses.ExternalTools#mxpost

```
[LM:nc=pos]
factors = "pos"
order = 7
settings = "-interpolate -unk"
raw-corpus = $wmt10-data/training/news-commentary10.$pair-extension.$output-extension
```

This factored model using all the available corpora is identical to the Edinburgh submission to the WMT 2010 shared task for English-Spanish, Spanish-English, and English-German language pairs (the French language pairs also used the 10^9 corpus, the Czech language pairs did not use the POS language model, and German-English used additional pre-processing steps).

Hierarchical model

Hierarchical phrase models allow for rules with gaps. Since these are represented by non-terminals and such rules are best processed with a search algorithm that is similar to syntactic chart parsing, such models fall into the class of tree-based or grammar-based models. For more information, please check the Syntax Tutorial (page **??**).

From the view of setting up hierarchical models with experiment.perl, very little has to be changed in comparison to the configuration file for phrase-based models:

```
% diff config.basic config.hierarchical
33c33
< decoder = $moses-src-dir/moses-cmd/src/moses
---
> decoder = $moses-src-dir/moses-chart-cmd/src/moses_chart
36c36
< ttable-binarizer = $moses-src-dir/misc/processPhraseTable
---
> #ttable-binarizer = $moses-src-dir/misc/processPhraseTable
39c39
< #ttable-binarizer = "$moses-src-dir/CreateOnDisk/src/CreateOnDiskPt 1 1 5 100 2"
---
> ttable-binarizer = "$moses-src-dir/CreateOnDisk/src/CreateOnDiskPt 1 1 5 100 2"
280c280
< lexicalized-reordering = msd-bidirectional-fe
---
> #lexicalized-reordering = msd-bidirectional-fe
284c284
< #hierarchical-rule-set = true
---
> hierarchical-rule-set = true
413c413
< decoder-settings = "-search-algorithm 1 -cube-pruning-pop-limit 5000 -s 5000"
---
> #decoder-settings = ""
```

The changes are: a different decoder binary (by default compiled into `moses-chart-cmd/src/moses_chart`) and `ttable-binarizer` are used. The decoder settings for phrasal cube pruning do not apply. Also, hierarchical models do not allow for lexicalized reordering (their rules fulfill the same purpose), and the setting for hierarchical rule sets has to be turned on. The use of hierarchical rules is indicated with the setting `hierarchical-rule-set`.

Target syntax model

Syntax models imply the use of linguistic annotation for the non-terminals of hierarchical models. This requires running a syntactic parser.

In our example `config.syntax`, syntax is used only on the English target side. The syntactic constituents are labeled with Collins parser, which needs to be installed first. Please read the installation instructions[56].

Compared to the hierarchical model, very little has to be changed in the configuration file:

```
% diff config.hierarchical config.syntax
46a47,49
> # syntactic parsers
> collins = /home/pkoehn/bin/COLLINS-PARSER
> output-parser = "$moses-script-dir/training/wrappers/parse-en-collins.perl"
>
241c244
< #extract-settings = ""
---
> extract-settings = "--MinHoleSource 1 --NonTermConsecSource"
```

The parser needs to be specified, and the extraction settings may be adjusted. And you are ready to go.

3.4.5 Try a Few More Things

Stemmed Word Alignment

The factored translation model training makes it very easy to set up word alignment not based on the surface form of words, but any other property of a word. One relatively popular method is to use stemmed words for word alignment.

There are two reasons for this: For one, for morphologically rich languages, stemming overcomes data sparsity problems. Secondly, GIZA++ may have difficulties with very large vocabulary sizes, and stemming reduces the number of unique words.

To set up stemmed word alignment in experiment.perl, you need to define a stem as a factor:

[56] http://www.statmt.org/moses/?n=Moses.ExternalTools#collins

```
[OUTPUT-FACTOR:stem4]
factor-script = "$moses-script-dir/training/wrappers/make-factor-stem.perl 4"

[INPUT-FACTOR:stem4]
factor-script = "$moses-script-dir/training/wrappers/make-factor-stem.perl 4"
```

and indicate the use of this factor in the `TRAINING` section:

```
input-factors = word stem4
output-factors = word stem4
alignment-factors = "stem4 -> stem4"
translation-factors = "word -> word"
reordering-factors = "word -> word"
#generation-factors =
decoding-steps = "t0"
```

Using Multi-Threaded GIZA++

GIZA++ is one of the slowest steps in the training pipeline. Qin Gao implemented a multi-threaded version of GIZA++, called MGIZA, which speeds up word alignment on multi-core machines.

To use MGIZA, you will first need to install[57] it.

To use it, you simply need to add some training options in the section `TRAINING`:

```
### general options
#
training-options = "-mgiza -mgiza-cpus 8"
```

Using Berkeley Aligner

The Berkeley Aligner is a alternative to GIZA++ for word alignment. You may (or may not) get better results using this tool.

To use the Berkeley Aligner, you will first need to install[58] it.

The example configuration file already has a section for the parameters for the tool. You need to un-comment them and adjust `berkeley-jar` to your installation. You should comment out `alignment-symmetrization-method`, since this is a GIZA++ setting.

[57] http://www.statmt.org/moses/?n=Moses.ExternalTools#mgiza

[58] http://www.statmt.org/moses/?n=Moses.ExternalTools#berkeley

```
### symmetrization method to obtain word alignments from giza output
# (commonly used: grow-diag-final-and)
#
#alignment-symmetrization-method = grow-diag-final-and

### use of berkeley aligner for word alignment
#
use-berkeley = true
alignment-symmetrization-method = berkeley
berkeley-train = $moses-script-dir/ems/support/berkeley-train.sh
berkeley-process =  $moses-script-dir/ems/support/berkeley-process.sh
berkeley-jar = /your/path/to/berkeleyaligner-2.1/berkeleyaligner.jar
berkeley-java-options = "-server -mx30000m -ea"
berkeley-training-options = "-Main.iters 5 5 -EMWordAligner.numThreads 8"
berkeley-process-options = "-EMWordAligner.numThreads 8"
berkeley-posterior = 0.5
```

The Berkeley Aligner proceeds in two step: a training step to learn the alignment model from the data and a processing step to find the best alignment for the training data. This step has the parameter `berkeley-posterior` to adjust a bias towards more or less alignment points. You can try different runs with different values for this parameter. Experiment.perl will not re-run the training step, just the processing step.

IRST Language Model

The provided examples use the SRI language model during decoding. When you want to use the IRSTLM instead, an additional processing step is required: the language model has to converted into a binary format.

This part of the `LM` section defines the use of IRSTLM:

```
### script to use for binary table format for irstlm
# (default: no binarization)
#
#lm-binarizer = $moses-src-dir/irstlm/bin/compile-lm

### script to create quantized language model format (irstlm)
# (default: no quantization)
#
#lm-quantizer = $moses-src-dir/irstlm/bin/quantize-lm
```

If you un-comment `lm-binarizer`, IRSTLM will be used. If you comment out in addition `lm-quantizer`, the language model will be compressed into a more compact representation.

Note that the values above assume that you installed the IRSTLM toolkit in the directory `$moses-src-dir/irstlm`.

Randomized Language Model

Randomized language models allow a much more compact (but lossy) representation. Being able to use much larger corpora for the language model may be beneficial over the small chance of making mistakes.

First of all, you need to install[59] the RandLM toolkit.

There are two different ways to train a randomized language model. One is to train it from scratch. The other way is to convert a SRI language model into randomized representation.

Training from scratch: Find the following section in the example configuration files and un-comment the `rlm-training` setting. Note that the section below assumes that you installed the randomized language model toolkit in the directory `$moses-src-dir/randlm`.

```
### tool to be used for training randomized language model from scratch
# (more commonly, a SRILM is trained)
#
rlm-training = "$moses-src-dir/randlm/bin/buildlm -falsepos 8 -values 8"
```

Converting SRI language model: Find the following section in the example configuration files and un-comment the `lm-randomizer` setting.

```
### script to use for converting into randomized table format
# (default: no randomization)
#
lm-randomizer = "$moses-src-dir/randlm/bin/buildlm -falsepos 8 -values 8"
```

You may want to try other values for `falsepos` and `values`. Please see the language model section on RandLM[60] for some more information about these parameters.

You can also randomize a interpolated language model by specifying the `lm-randomizer` in the section `INTERPOLATED-LM`.

Compound Splitting

Compounding languages, such as German, allow the creation of long words such as *Neuwortgenerierung* (*new word generation*). This results in a lot of unknown words in any text, so splitting up these compounds is a common method when translating from such languages.

[59] http://www.statmt.org/moses/?n=FactoredTraining.BuildingLanguageModel#randlm
[60] http://www.statmt.org/moses/?n=FactoredTraining.BuildingLanguageModel#randlm

Moses offers a support tool that splits up words, if the geometric average of the frequency of its parts is higher than the frequency of a word. The method requires a model (the frequency statistics of words in a corpus), so there is a training and application step.

Such word splitting can be added to experiment.perl simply by specifying the splitter script in the GENERAL section:

```
input-splitter = $moses-script-dir/generic/compound-splitter.perl
```

Splitting words on the output side is currently not supported.

3.4.6 A Short Manual

The basic lay of the land is: experiment.perl breaks up the training, tuning, and evaluating of a statistical machine translation system into a number of steps, which are then scheduled to run in parallel or sequence depending on their inter-dependencies and available resources. The possible steps are defined in the file `experiment.meta`. An experiment is defined by a configuration file.

The main modules of running an experiment are:

- `CORPUS`: preparing a parallel corpus,
- `INPUT-FACTOR` and `OUTPUT-FACTOR`: commands to create factors,
- `TRAINING`: training a translation model,
- `LM`: training a language model,
- `INTERPOLATED-LM`: interpolate language models,
- `SPLITTER`: training a word splitting model,
- `RECASING`: training a recaser,
- `TRUECASING`: training a truecaser,
- `TUNING`: running minumum error rate training to set component weights,
- `TESTING`: translating and scoring a test set, and
- `REPORTING`: compile all scores in one file.

Experiment.Meta

The actual steps, their dependencies and other salient information are to be found in the file `experiment.meta`. Think of experiment.meta as a "template" file.

Here the parts of the step description for `CORPUS:get-corpus` and `CORPUS:tokenize`:

```
get-corpus
in: get-corpus-script
out: raw-stem
[...]

tokenize
```

```
in: raw-stem
out: tokenized-stem
[...]
```

Each step takes some input (`in`) and provides some output (`out`). This also establishes the dependencies between the steps. The step `tokenize` requires the input `raw-stem`. This is provided by the step `get-corpus`.

`experiment.meta` provides a generic template for steps and their interaction. For an actual experiment, a configuration file determines which steps need to be run. This configuration file is the one that is specified when invocing `experiment.perl`. It may contain for instance the following:

```

[CORPUS:europarl]

### raw corpus files (untokenized, but sentence aligned)
#
raw-stem = $europarl-v3/training/europarl-v3.fr-en
```

Here, the parallel corpus to be used is named `europarl` and it is provided in raw text format in the location `$europarl-v3/training/europarl-v3.fr-en` (the variable `$europarl-v3` is defined elsewhere in the config file). The effect of this specification in the config file is that the step `get-corpus` does not need to be run, since its output is given as a file. More on the configuration file below in the next section.

Several types of information are specified in `experiment.meta`:

- `in` and `out`: Established dependencies between steps; input may also be provided by files specified in the configuration.
- `default-name`: Name of the file in which the output of the step will be stored.
- `template`: Template for the command that is placed in the execution script for the step.
- `template-if`: Potential command for the execution script. Only used, if the first parameter exists.
- `error`: `experiment.perl` detects if a step failed by scanning STDERR for key words such as killed, error, died, not found, and so on. Additional key words and phrase are provided with this parameter.
- `not-error`: Declares default error key words as not indicating failures.
- `pass-unless`: Only if the given parameter is defined, this step is executed, otherwise the step is passed (illustrated by a yellow box in the graph).
- `ignore-unless`: If the given parameter is defined, this step is not executed. This overrides requirements of downstream steps.
- `rerun-on-change`: If similar experiments are run, the output of steps may be used, if input and parameter settings are the same. This specifies a number of parameters whose change disallows a re-use in different run.
- `parallelizable`: When running on the cluster, this step may be parallelized (only if `generic-parallelizer` is set in the config file, typically to the script `$edinburgh-script-dir/generic-par`

- qsub-script: If running on a cluster, this step is run on the head node, and not submitted to the queue (because it submits jobs itself).

Here now the full definition of the step CONFIG:tokenize

```
tokenize
in: raw-stem
out: tokenized-stem
default-name: corpus/tok
pass-unless: input-tokenizer output-tokenizer
template-if: input-tokenizer IN.$input-extension OUT.$input-extension
template-if: output-tokenizer IN.$output-extension OUT.$output-extension
parallelizable: yes
```

The step takes raw-stem and produces tokenized-stem. It is parallizable with the generic parallelizer.

That output is stored in the file corpus/tok. Note that the actual file name also contains the corpus name, and the run number. Also, in this case, the parallel corpus is stored in two files, so file name may be something like corpus/europarl.tok.1.fr and corpus/europarl.tok.1.en.

The step is only executed, if either input-tokenizer or output-tokenizer are specified. The templates indicate how the command lines in the execution script for the steps look like.

Multiple Corpora, One Translation Model

We may use multiple parallel corpora for training a translation model or multiple monolingual corpora for training a language model. Each of these have their own instances of the CORPUS and LM module. There may be also multiple test sets in TESTING). However, there is only one translation model and hence only one instance of the TRAINING module.

The definitions in experiment.meta reflect the different nature of these modules. For instance CORPUS is flagged as multiple, while TRAINING is flagged as single.

When defining settings for the different modules, the singular module TRAINING has only one section, while this one general section and specific LM sections for each training corpus. In the specific section, the corpus is named, e.g. LM:europarl.

As you may imagine, the tracking of dependencies between steps of different types of modules and the consolidation of corpus-specific instances of modules is a bit complex. But most of that is hidden from the user of the Experimental Management System.

When looking up the parameter settings for a step, first the set-specific section (LM:europarl) is consulted. If there is no definition, then the module definition (LM) and finally the general definition (in section GENERAL) is consulted. In other words, local settings override global settings.

Defining Settings

The configuration file for experimental runs is a collection of parameter settings, one per line with empty lines and comment lines for better readability, organized in sections for each of the modules.

The syntax of setting definition is `setting = value` (note: spaces around the equal sign). If the value contains spaces, it must be placed into quotes (`setting = "the value"`), except when a vector of values is implied (only used when defining list of factors: `output-factor = word pos`.

Comments are indicated by a hash (#).

The start of sections is indicated by the section name in square brackets (`[TRAINING]` or `[CORPUS:europarl]`). If the word `IGNORE` is appended to a section definition, then the entire section is ignored.

Settings can be used as variables to define other settings:

```
working-dir = /home/pkoehn/experiment
wmt10-data = $working-dir/data
```

Variable names may be placed in curly brackets for clearer separation:

```
wmt10-data = ${working-dir}/data
```

Such variable references may also reach other modules:

```
[RECASING]
tokenized = $LM:europarl:tokenized-corpus
```

Finally, reference can be made to settings that are not defined in the configuration file, but are the product of the defined sequence of steps.

Say, in the above example, `tokenized-corpus` is not defined in the section `LM:europarl`, but instead `raw-corpus`. Then, the tokenized corpus is produced by the normal processing pipeline. Such an intermediate file can be used elsewhere:

```
[RECASING]
tokenized = [LM:europarl:tokenized-corpus]
```

Some error checking is done on the validity of the values. All values that seem to be file paths trigger the existence check for such files. A file with the prefix of the value must exist.

There are a lot of settings reflecting the many steps, and explaining these would require explaining the entire training, tuning, and testing pipeline. Please find the required documentation for step elsewhere around here. Every effort has been made to include verbose descriptions in the example configuration files, which should be taken as starting point.

Working with Experiment.Perl

You have to define an experiment in a configuration file and the Experiment Management System figures out which steps need to be run and schedules them either as jobs on a cluster or runs them serially on a single machine.

Other options:

- `-no-graph`: Supresses the display of the graph.
- `-continue RUN`: Continues the experiment `RUN`, which crashed earlier. Make sure that crashed step and its output is deleted (see more below).
- `-max-active`: Specifies the number of steps that can be run in parallel when running on a single machine (default: 2, not used when run on cluster).
- `-sleep`: Sets the number of seconds to be waited in the scheduler before the completion of tasks is checked (default: 2).
- `-ignore-time`: Changes the re-use behavior. By default files cannot be re-used when their time stamp changed (typically a tool such as the tokenizer which was changed, thus requiring re-running all tokenization steps in new experiments). With this switch, files with changed time stamp can be re-used.
- `-meta`: Allows the specification of a custom `experiment.meta` file, instead of using the one in the same directory as the `experiment.perl` script.
- `-cluster`: Indicates that the current machine is a cluster head node. Step files are submitted as jobs to the cluster.
- `-multicore`: Indicates that the current machine is a multi-core machine. This allows for additional parallelization with the generic parallelizer setting.

The script may automatically detect if it is run on a compute cluster or a multi-core machine, if this is specified in the file `experiment.machines`, for instance:

```
cluster: townhill seville
multicore-8: tyr thor
multicore-16: loki
```

defines the machines `townhill` and `seville` as GridEngine cluster machines, `tyr` and `thor` as 8-core machines and `loki` as 16-core machines.

Typically, experiments are started with the command:

```
experiment.perl -config my-config -exec
```

Since experiments run for a long time, you may want to run this in the background and also set a nicer priority:

```
nice nohup -config my-config -exec >& OUT.[RUN] &
```

This keeps also a report (STDERR and STDOUT) on the execution in a file named, say, `OUT.1`, with the number corresponding to the run number.

The meta-information for the run is stored in the directory `steps`. Each run has a sub directory with its number (`steps/1`, `steps/2`, etc.). The sub directory `steps/0` contains step specification when Experiment.perl is called without the `-exec` switch.

The sub directories for each run contain the step definitions, as well as their meta-information and output. The sub directories also contain a copy of the configuration file (e.g. `steps/1/config.1`), the agenda graph (e.g. `steps/1/graph.1.{dot,ps,png}`), a file containing all expanded parameter settings (e.g. `steps/1/parameter.1`), and an empty file that is touched every minute as long as the experiment is still running (e.g. `steps/1/running.1`).

Continuing Crashed Experiments

Steps may crash. No, steps will crash, be it because faulty settings, faulty tools, problems with the computing resources, willful interruption of an experiment, or an act of God.

The first thing to continue a crashed experiment is to detect the crashed step. This is shown either by the red node in the displayed graph or reported on the command line in the last lines before crashing; though this may not be pretty obvious, if parallel steps kept running after that. However, the automatic error detection is not perfect and a step may have failed upstream without detection causes failure further down the road.

You should have a understanding of what each step does. Then, by looking at its `STDERR` and `STDOUT` file, and the output files it should have produced, you can track down what went wrong.

Fix the problem, and delete all files associated with the failed step (e.g., `rm steps/13/TUNING_tune.13*`, `rm -r tuning/tmp.1`). To find what has been produced by the crashed step, you may need to consult where the output of this step is placed, by looking at `experiment.meta`.

You can continue a crashed experimental run (e.g., run number 13) with:

```
experiment.perl -continue 13 -exec
```

You may want to check what will be run by excluding the `-exec` command at first. The graph indicates which steps will be re-used from the original crashed run.

If the mistake was a parameter setting, you can change that setting in the stored configuration file (e.g., `steps/1/config.1`). Take care, however, to delete all steps (and their subsequent steps) that would have been run differently with that setting.

If an experimental run crashed early, or you do not want to repeat it, it may be easier to delete the entire step directory (`rm -r steps/13`). Only do this with the latest experimental run (e.g., not when there is already a run 14), otherwise it may mess up the re-use of results.

You may also delete all output associated with a run with the command `rm -r */*.13*`. However this requires some care, so you may want to check first what you are deleting (`ls */*.13`).

Running on a Cluster

Experiment.perl works with Sun GridEngine clusters. The script needs to be run on the head node and jobs are scheduled on the nodes.

There are two ways to tell experiment.perl that the current machine is a cluster computer. One is by using the switch `-cluster`, or by adding the machine name into `experiment.machines`.

The configuration file has a section that allows for the setting of cluster-specific settings. The setting `jobs` is used to specify into how many jobs to split the decoding during tuning and testing. For more details on this, please see `moses-parallel.pl`.

All other settings specify switches that are passed along with each submission of a job via qsub:

- `qsub-memory`: number of memory slots (`-pe memory NUMBER`),
- `qsub-hours`: number of hours reserved for each job (`-l h_rt=NUMBER:0:0`),
- `qsub-project`: name if the project for user accounting (`-P PROJECT`), and
- `qsub-settings`: any other setting that is passed along verbatim.

Note that the general settings can be overriden in each module definition - you may want to have different settings for different steps.

If the setting `generic-parallelizer` is set (most often it is set to to the ems support script `$moses-script-dir/ems/support/generic-parallelizer.perl`), then a number of additional steps are parallelized. For instance, tokenization is performed by breaking up the corpus into as many parts as specified with `jobs`, jobs to process the parts are submitted in parallel to the cluster, and their output pieced together upon completion.

Be aware that there are many different ways to configure a GridEngine cluster. Not all the options described here may be available, and it my not work out of the box, due to your specific installation.

Running on a Multi-core Machine

Using a multi-core machine means first of all that more steps can be scheduled in parallel. There is also a generic parallelizer (`generic-multicore-parallelizer.perl`) that plays the same role as the generic parallelizer for clusters.

However, decoding is not broken up into several pieces. It is more sensible to use multi-threading in the decoder[61].

Web Interface

The introduction included some screen shots of the web interface to the Experimental Management System. You will need to have a running web server on a machine (LAMPP on Linux or MAMP on Mac does the trick) that has access to the file system where your working directory is stored.

Copy or link the `web` directory (in `scripts/ems`) on a web server. Make sure the web server user has the right write permissions on the web interface directory.

To add your experiments to this interface, add a line to the file `/your/web/interface/dir/setup`. The format of the file is explained in the file.

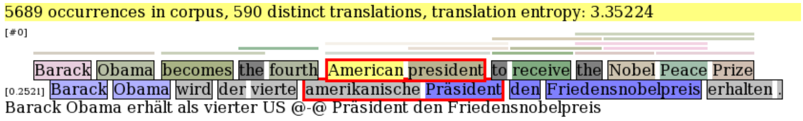
[61] http://www.statmt.org/moses/?n=Moses.AdvancedFeatures#multi-threaded

3.4.7 Analysis

You can include additional analysis for an experimental run into the web interface by specifying the setting `analysis` in its configuration file.

```
analysis = $moses-script-dir/ems/support/analysis.perl
```

This currently reports n-gram precision and recall statistics and color-coded n-gram correctness markup for the output sentences, as in



5689 occurrences in corpus, 590 distinct translations, translation entropy: 3.35224

[#0]

Barack Obama becomes the fourth American president to receive the Nobel Peace Prize

[0.2521] Barack Obama wird der vierte amerikanische Präsident den Friedensnobelpreis erhalten .

Barack Obama erhält als vierter US @-@ Präsident den Friedensnobelpreis

The output is color-highlighted according to n-gram matches with the reference translation. The following colors are used:

- grey: word not in reference,
- light blue: word part of 1-gram match,
- blue: word part of 2-gram match,
- dark blue: word part of 3-gram match, and
- very dark blue: word part of 4-gram match.

Segmentation

The setting `analyze-coverage` include a coverage analysis: which words and phrases in the input occur in the training data or the translation table? This is reported in color coding and in a yellow report box when moving the mouse of the word or the phrase. Also, summary statistics for how many words occur how often are given, and a report on unknown or rare words is generated.

Coverage Analysis

The setting `analyze-coverage` include a coverage analysis: which words and phrases in the input occur in the training data or the translation table? This is reported in color coding and in a yellow report box when moving the mouse of the word or the phrase. Also, summary statistics for how many words occur how often are given, and a report on unknown or rare words is generated.

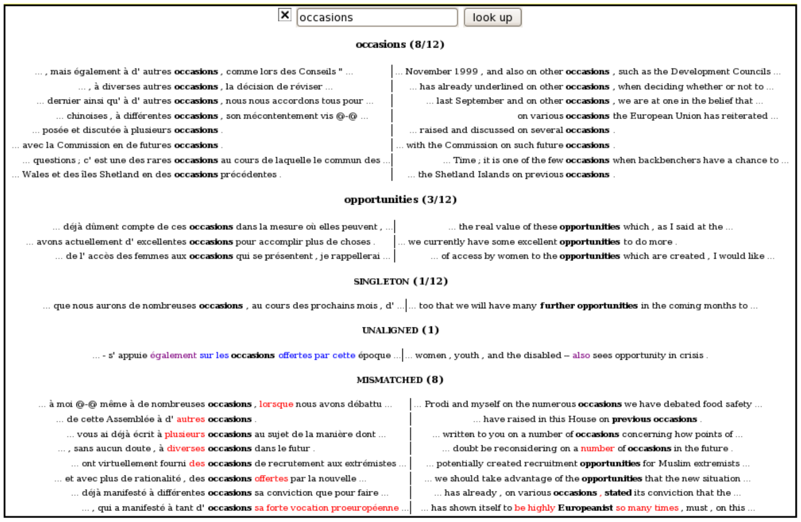
Bilingual Concordancer

To more closely inspect where input words and phrases occur in the training corpus, the analysis tool includes a bilingual concordancer. You turn it on by adding this line to the training section of your configuration file:

```
biconcor = $moses-script-dir/ems/biconcor/biconcor
```

During training, a suffix array of the corpus is built in the model directory. The analysis web interface accesses these binary files to quickly scan for occurrences of source words and phrases in the training corpus. For this to work, you need to include the biconcor binary in the web root directory.

When you click on a word or phrase, the web page is augemented with a section that shows all (or frequent word, a sample of all) occurences of that phrase in the corpus, and how it was aligned:



occasions (8/12)

... , mais également à d' autres **occasions** , comme lors des Conseils " ...	... November 1999 , and also on other **occasions** , such as the Development Councils ...
... , à diverses autres **occasions** , la décision de réviser ...	... has already underlined on other **occasions** , when deciding whether or not to ...
... dernier ainsi qu' à d' autres **occasions** , nous nous accordons tous pour ...	... last September and on other **occasions** , we are at one in the belief that ...
... chinoises , à différentes **occasions** , son mécontentement vis @-@ ...	on various **occasions** the European Union has reiterated ...
... posée et discutée à plusieurs **occasions** .	... raised and discussed on several **occasions** .
... avec la Commission en de futures **occasions** .	... with the Commission on such future **occasions** .
... questions ; c' est une des rares **occasions** au cours de laquelle le commun des ...	... Time ; it is one of the few **occasions** when backbenchers have a chance to ...
... Wales et des îles Shetland en des **occasions** précédentes .	... the Shetland Islands on previous **occasions** .

opportunities (3/12)

... déjà dûment compte de ces **occasions** dans la mesure où elles peuvent , ...	... the real value of these **opportunities** which , as I said at the ...
... avons actuellement d' excellentes **occasions** pour accomplir plus de choses .	... we currently have some excellent **opportunities** to do more .
... de l' accès des femmes aux **occasions** qui se présentent , je rappellerai ...	... of access by women to the **opportunities** which are created , I would like ...

SINGLETON (1/12)

... que nous aurons de nombreuses **occasions** , au cours des prochains mois , d' ...	... too that we will have many **further opportunities** in the coming months to ...

UNALIGNED (1)

... - s' appuie également sur les **occasions** offertes par cette époque ...	... women , youth , and the disabled – also sees opportunity in crisis .

MISMATCHED (8)

... à moi @-@ même à de nombreuses **occasions** , lorsque nous avons débattu ...	... Prodi and myself on the numerous **occasions** we have debated food safety ...
... de cette Assemblée à d' autres **occasions** .	... have raised in this House on **previous occasions** .
... vous ai déjà écrit à plusieurs **occasions** au sujet de la manière dont ...	... written to you on a number of **occasions** concerning how points of ...
... , sans aucun doute , à diverses **occasions** dans le futur .	... doubt be reconsidering on a number of **occasions** in the future .
... ont virtuellement fourni des **occasions** de recrutement aux extrémistes ...	... potentially created recruitment **opportunities** for Muslim extremists ...
... et avec plus de rationalité , des **occasions** offertes par la nouvelle ...	... we should take advantage of the **opportunities** that the new situation ...
... déjà manifesté à différentes **occasions** sa conviction que pour faire ...	... has already , on various **occasions** , **stated** its conviction that the ...
... , qui a manifesté à tant d' **occasions** sa forte vocation proeuropéenne ...	... has shown itself to be highly **Europeanist** so many times , must , on this ...

Source occurrences (with context) are shown on the left half, the aligned target on the right.

In the main part, occurrences are grouped by different translations — also shown bold in context. Unaligned boundary words are shown in blue. The extraction heuristic extracts additional rules for these cases, but these are not listed here for clarity.

At the end, source occurrences for which no rules could be extracted are shown. This may happen because the source words are not aligned to any target words. In this case, the tool shows alignments of the previous word (purple) and following word(olive), as well as some neighboring unaligned words (again, in blue). Another reason for failure to extract rules are misalignments, when the source phrase maps to a target span which contains words that also align to outside source words (violation of the coherence contraint). These misaligned words (in source and target) are shown in red.

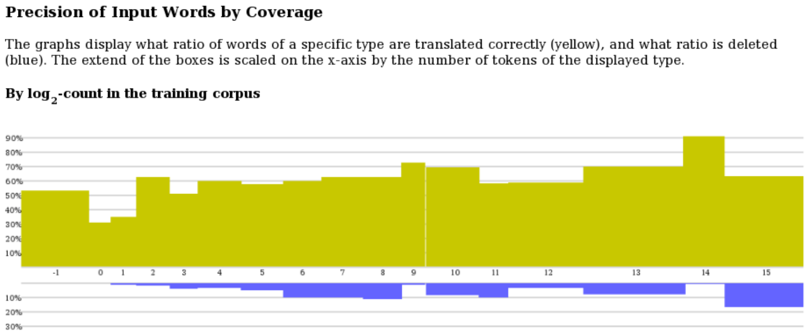
Precision by coverage

To investigate further, if the correctness of the translation of input words depends on frequency in the corpus (and what the distribution of word frequency is), a report for precision by coverage can be turned on with the following settings:

```
report-precision-by-coverage = yes
precision-by-coverage-factor = pos
precision-by-coverage-base = $working-dir/evaluation/test.analysis.5
```

Only the first setting `report-precision-by-coverage` is needed for the report. The second setting `precision-by-coverage-factor` provides an additional breakdown for a specific input factor (in the example, the part-of-speech factor named `pos`). More on the `precision-by-coverage-base` below.

When clicking on "precision of input by coverage" on the main page, a precision by coverage graph is shown:



Precision of Input Words by Coverage

The graphs display what ratio of words of a specific type are translated correctly (yellow), and what ratio is deleted (blue). The extend of the boxes is scaled on the x-axis by the number of tokens of the displayed type.

By $\log_2$-count in the training corpus

The log-coverage class is on the x-axis (-1 meaning unknown, 0 singletons, 1 words that occur twice, 2 words that occur 3-4 times, 3 words that occur 5-8 times, and so on). The scale of boxes for each class is determined by the ratio of words in the class in the test set. The precision of translations of words in a class is shown on the y-axis.

Translation of precision of input words cannot be determined in a clear cut word. Our determination relies on phrase alignment of the decoder, word alignment within phrases, and accounting for multiple occurrences of transled words in output and reference translations. Not that the precision metric does not penalize for dropping words, so this is shown in a second graph (in blue), below the precision graph.

If you click on the graph, you will see the graph in tabular form. Following additional links allows you to see breakdowns for the actual words, and even find the sentences in which they occur.

Finally, the `precision-by-coverage-base` setting. For comparison purposes, it may be useful to base the coverage statistics on the corpus of a previous run. For instance, if you add training data, does the translation quality of the words increase? Well, a word that occured 3 times in the small corpus, may now occur 10 times in the big corpus, hence the word is placed in a different class. To maintain the original classification of words into the log-coverage classes, you may use this setting to point to an earlier run.

Subsection last modified on April 25, 2012, at 07:48 PM

3.5 Translating Web pages with Moses

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

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.

3.5.1 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.

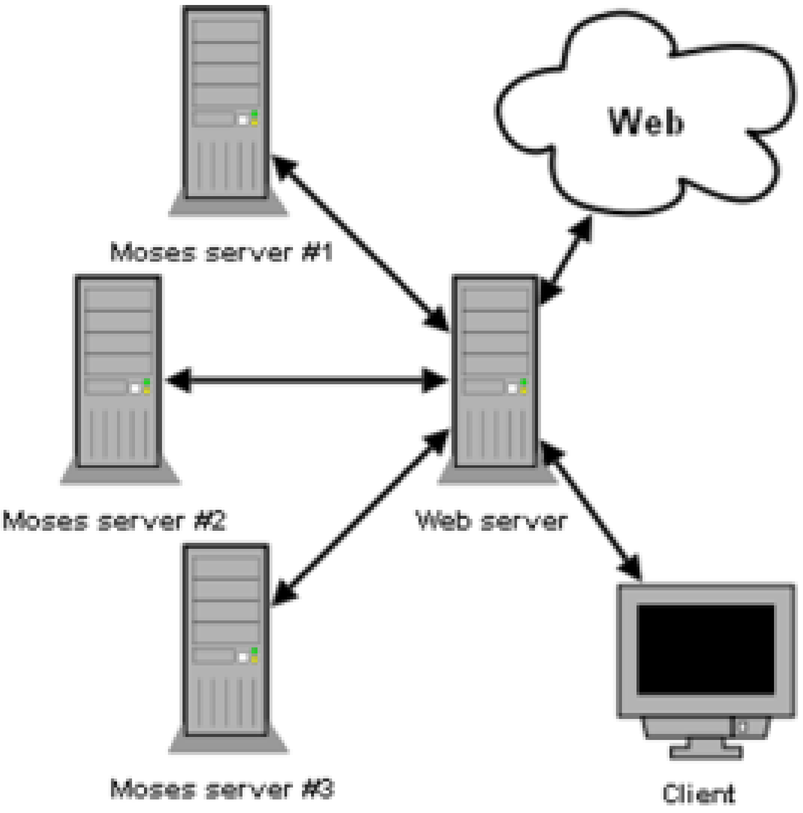
Before starting, the reader should ideally possess basic knowledge of:

- 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 resposi-tory, 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.

3.5.2 Detailed setup instructions

Obtaining a copy of the scripts

The scripts are stored in the contrib/web directory in the Moses distribution.

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 network. It was used to start and stop the Moses servers all at once, also setting up SSH tunneling 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 receives 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`

Subsection last modified on April 25, 2012, at 09:04 PM

4

Training Manual

4.1 Training

4.1.1 Content

- Training process (page **??**)
- Running the training script (page **??**)

4.1.2 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-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.

4.1.3 Running the training script

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

Run the training script:

```
train-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 **??**) at the end of this manual. For corpus preparation, see the section on how to prepare training data (page **??**).

Subsection last modified on May 04, 2010, at 10:05 PM

4.2 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`)

4.2.1 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
```

4.2.2 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`.

Subsection last modified on July 14, 2006, at 01:07 AM

4.3 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.

4.3.1 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
```

4.3.2 Reordering factors

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

4.3.3 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.

4.3.4 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.)

It is possible to specify multiple decoding paths, for instance by `-decoding-steps t0,g0,t1,t2,g1:t3`, where colons separate the paths. Translation options are generated from each decoding path and used during decoding.

Subsection last modified on July 03, 2012, at 05:16 AM

4.4 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
```

Subsection last modified on July 14, 2006, at 01:07 AM

4.5 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 })
```

4.5.1 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.

4.5.2 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!)

Subsection last modified on December 22, 2008, at 12:21 AM

4.6 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)

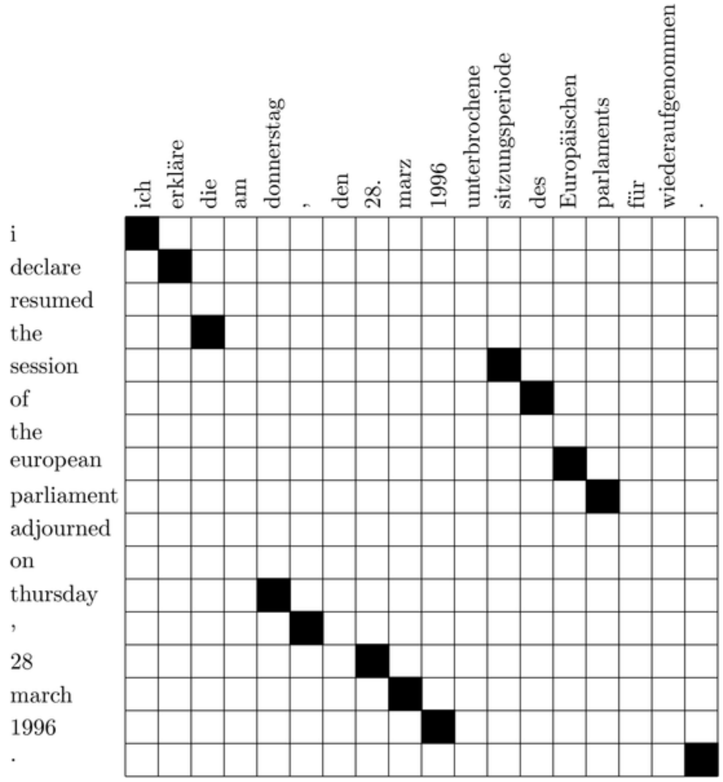
Alternative alignment methods can be specified with the switch `-alignment`.

Here, the pseudo code for the default heuristic:

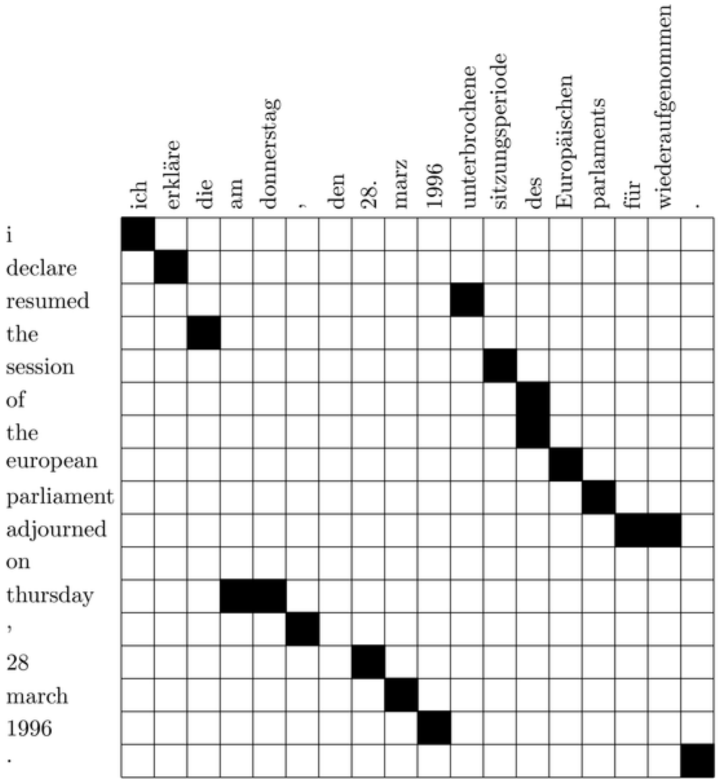
```
GROW-DIAG-FINAL(e2f,f2e):
neighboring = ((-1,0),(0,-1),(1,0),(0,1),(-1,-1),(-1,1),(1,-1),(1,1))
alignment = intersect(e2f,f2e);
GROW-DIAG(); FINAL(e2f); FINAL(f2e);

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 with the intersection of the two alignments for the second sentence in the corpus above



and then add some additional alignment points that lie in the union of the two alignments



	ich	erkläre	die	am	donnerstag	,	den	28.	marz	1996	unterbrochene	sitzungsperiode	des	Europäischen	parlaments	für	wiederaufgenommen	.
i	■																	
declare		■																
resumed											■							
the			■															
session												■						
of													■					
the													■					
european														■				
parliament															■			
adjourned																■	■	
on																		
thursday				■	■													
,						■												
28								■										
march									■									
1996										■								
.																		■

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.

Subsection last modified on April 26, 2012, at 05:17 PM

4.7 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
```

Subsection last modified on July 14, 2006, at 01:15 AM

4.8 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`.

Subsection last modified on July 14, 2006, at 01:15 AM

4.9 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:

1. inverse phrase translation probability $\phi(f|e)$
2. inverse lexical weighting $lex(f|e)$
3. direct phrase translation probability $\phi(e|f)$
4. direct lexical weighting $lex(e|f)$
5. phrase penalty (always $exp(1) = 2.718$)

Using a subset of scores

You may not want to use all the scores in your translation table. The following options allow you to remove some of the scores:

- `NoLex` – do not use lexical scores (removes score 2 and 4)
- `OnlyDirect` – do not use the inverse scores (removes score 1 and 2)
- `NoPhraseCount` – do not use the phrase count feature (removes score 5)

These settings have to be specified with the setting `-score-options` when calling the script `train-model.perl`, for instance:

```
train-model.perl [... other settings ...] -score-options ’--NoLex’
```

Good Turing discounting

Singleton phrase pairs tend to have overestimated phrase translation probabilities. Consider the extreme case of a source phrase that occurs only once in the corpus and has only one translation. The corresponding phrase translation probability $\phi(e|f)$ would be 1.

To obtain better phrase translation probabilities, the observed counts may be reduced by *expected* counts which takes unobserved events into account. Borrowing a method from language model estimation, Good Turing discounting can be used to reduce the actual counts (such as 1 in the example above) and reduce it to a more realistic number (maybe 0.3). The value of the adjusted count is determined by an analysis of the number of singleton, twice-occuring, thrice-occuring, etc. phrase pairs that were extracted.

To use Good Turing discounting of the phrase translation probabilities, you have to specify `-GoodTuring` as one of the `-score-options`, as in the section above. The adjusted counts are reported to STDERR.

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 **??**).

The alignments information are reported in the fourth fields. The format is identical to the alignment output obtained when the GIZA++ output has been symmetrized priot to phrase extraction.

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

For instance:

```
in der europaeischen union ||| in europe ||| 0-0 2-1 3-1 ||| ...
```

means

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

The word-to-word alignments come from one word alignment (see training step "Align words" (page **??**)).

The alignment information is also used in SCFG-rules for the chart-decoder to link non-terminals together in the source and target side. In this instance, the alignment information is not an option, but a necessity. For example, the following Moses SCFG rule

```
[X][X] miss [X][X] [X] ||| [X][X] [X][X] manques [X] ||| ... ||| 0-1 2-0 ||| ...
```

is formated as this in the Hiero format:

```
[X] ||| [X,1] miss [X,2] ||| [X,2] [X,1] manques ||| ....
```

ie. this rule reordes the 1st and 3rd non-terminals in the source.

Therefore, the same alignment field can be used for word-alignment and non-terminal co-indexes. However, I'm (Hieu) sure if anyone has implemented this in the chart decoder yet

Subsection last modified on June 11, 2011, at 05:36 AM

4.10 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, so called lexicalized reordering models, may be build. There are three types of lexicalized reordering models in Moses that are based on Koehn et al., 2005[1] and Galley and Manning, 2008[2]. The Koehn at al. model determines the orientation of two phrases based on word alignments at training time, and based on phrase alignments at decoding time. The other two models are based on Galley and Manning. The phrase-based model uses phrases both at training and decoding time, and the hierarchical model allows combinations of several phrases for determining the orientation.

The lexicalized reordering models are specified by a configuration string, containing five parts, that account for different aspects:

- **Modeltype** - the type of model used (see above)
  - wbe - word-based extraction (but phrase-based at decoding). This is the original model in Moses. **DEFAULT**
  - phrase - phrase-based model
  - hier - hierarchical model
- **Orientation** - Which classes of orientations that are used in the model
  - mslr - Considers four different orientations: `monotone`, `swap`, `discontinuous-left`, `discontinuous-right`
  - msd - Considers three different orientations: `monotone`, `swap`, `discontinuous` (the two discontinuous classes of the mslr model are merged into one class)
  - monotonicity - Considers two different orientations: `monotone` or `non-monotone` (`swap` and `discontinuous` of the msd model are merged into the `non-monotone` class)
  - leftright - Considers two different orientations: `left` or `right` (the four classes in the mslr model are merged into two classes, `swap` and `discontinuous-left` into `left` and `monotone` and `discontinuous-right` into `right`)
- **Directionality** - Determines if the orientation should be modeled based on the previous or next phrase, or both.
  - backward - determine orientation with respect to previous phrase **DEFAULT**
  - forward - determine orientation with respect to following phrase
  - bidirectional - use both backward and forward models
- **language** - decides which language to base the model on
  - fe - conditioned on both the source and target languages
  - f - conditioned on the source language only
- **collapsing** - determines how to treat the scores
  - allff - treat the scores as individual feature functions **DEFAULT**
  - collapseff - collapse all scores in one direction into one feature function

any possible configuation of these five factors is allowed. It is always necessary to specify orientation and language. The other three factors use the default values indicated above if they are not specified. Some examples of possible models are:

[1] http://homepages.inf.ed.ac.uk/pkoehn/publications/iwslt05-report.pdf

[2] http://www.aclweb.org/anthology/D/D08/D08-1089.pdf

- `msd-bidirectional-fe` (this model is commonly used, for instance it is the model used in the WMT baselines[3])
- `wbe-msd-bidirectional-fe-allff` same model as above
- `mslr-f`
- `wbe-backward-mslr-f-allff` same model as above
- `phrase-msd-bidirectional-fe`
- `hier-mslr-bidirectional-fe`
- `hier-leftright-forward-f-collapseff`

and of course `distance`.

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

```
-reordering distance
-reordering msd-bidirectional-fe
-reordering msd-bidirectional-fe,hier-mslr-bidirectional-fe
-reordering distance,msd-bidirectional-fe,hier-mslr-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. If the flag `allff` is used, a `msd` model has three features, one each for the probability that the phrase is translated monotone, swapped, or discontinuous, a `mslr` model has four features and a `monotonicity` or `leftright` model has two features. If a `bidrectional` model is used, then the number of features doubles - one for each direction. If `collapseff` are used there is one feature for each direction, regardless of which orientation types that are used.

There are also a number of other flags that can be given to `train-model.perl` that concerns the reordering models:

- –reordering-smooth - specifies the smoothing constant to be used for training lexicalized reordering models. If the letter "u" follows the constant, smoothing is based on actual counts. (default 0.5)
- –max-lexical-reordering - if this flag is used, the extract file will contain information for the mslr orientations for all three model types, wbe, phrase and hier. Otherwise the extract file will contain the minimum information that is needed based on which reordering model config strings that are given.

Subsection last modified on March 27, 2012, at 08:57 AM

[3] http://www.statmt.org/wmt11/baseline.html

4.11 Training Step 8: Build generation model

The generation model is build from the target side of the parallel corpus.

By default, forward and backward probabilities are computed. If you use the switch `-generation-type single` only the probabilities in the direction of the step are computed.

Subsection last modified on May 05, 2010, at 06:00 PM

4.12 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.

Note that the configuration file set –by default– the usage of SRILM as a LM toolkit. If you prefer to use another LM toolkit, change the configuration file as described here[4]

Subsection last modified on September 26, 2011, at 09:16 AM

4.13 Building a Language Model

4.13.1 Content

- Language Models in Moses (page **??**)
- Building a LM with the SRILM Toolkit (page **??**)
- On the IRSTLM Toolkit (page **??**)
- Building Huge Language Models (page **??**)
- Binary Language Models (page **??**)
- Quantized Language Models (page **??**)
- Memory Mapping (page **??**)
- Class Language Models and more (page **??**)
- Chunk Language Models (page **??**)
- RandLM (page **??**)
- Installing RandLM (page **??**)
- Building a randomized language model (page **??**)

[4] http://www.statmt.org/moses/?n=FactoredTraining.BuildingLanguageModel#ntoc1

- Example 1: Building directly from corpora (page **??**)
- Example 2: Building from an ARPA file (SRILM) (page **??**)
- Example 3: Building a second randomized language model from the same data (page **??**)
- Building Randomised LMs from 100 Billion Words using Hadoop (page **??**)
- Querying Randomised Language Models (page **??**)
- KenLM (page **??**)
- Binary file (page **??**)
- Full or lazy loading (page **??**)
- Probing (page **??**)
- Trie (page **??**)
- Quantization (page **??**)
- Array compression (aka Chop) (page **??**)
- Vocabulary lookup (page **??**)

4.13.2 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[5], which is freely available.
- the IRST language modeling toolkit[6], which is freely available and open source.
- the RandLM language modeling toolkit[7], which is freely available and open source.
- the KenLM language modeling toolkit[8], which is included in Moses by default.

To use these language models, they have to be compiled with the proper option:

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

KenLM is compiled by default. In the Moses configuration file, the type (SRI/IRST/RandLM/KenLM) of the LM is specified through the first field (0/1/5/8 respectively) of the lines devoted to the specification of LMs:

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

[5] http://www.speech.sri.com/projects/srilm/

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

[7] http://sourceforge.net/projects/randlm/

[8] http://kheafield.com/code/kenlm/

or

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

or

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

or

```
8 <factor> <size> filename.arpa
```

The toolkits, except for KenLM, come with programs that create a language model file, as required by our decoder. KenLM reads ARPA files produced by SRI or IRST. IRST toolkit also reads the language model file produced by SRI.

4.13.3 Building a LM with the SRILM 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`.

4.13.4 On the IRSTLM Toolkit

Moses can also use language models created with the IRSTLM toolkit (see Federico & Cettolo, (ACL WS-SMT, 2007)[9]). The commands described in the following are supplied with the IRSTLM toolkit that has to be downloaded[10] 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:

[9] http://www.aclweb.org/anthology-new/W/W07/W07-0712.pdf

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

- building (huge) LMs (page **??**)
- quantizing LMs (page **??**)
- compiling LMs (possibly quantized) into a binary format (page **??**)
- accessing binary LMs through the memory mapping mechanism (page **??**)
- query class and chunk LMs (page **??**)

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[11].

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:

[11] 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)[12]) 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

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

```
language-model.blm
```

Rename it for adding the *.mm* suffix:

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

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.

4.13.5 RandLM

If you really want to build the largest LMs possible (for example, a 5-gram trained on one hundred billion words 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. RandLM is multithreaded now, so the speed reduction should be less of a problem.

Technical details of randomized language modelling can be found in a ACL paper (see Talbot and Osborne, (ACL 2007)[13])

Installing RandLM

RandLM is available at Sourceforge[14].

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. RandLM has supporting Hadoop scripts).

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:

[13] http://aclweb.org/anthology-new/P/P07/P07-1065.pdf

[14] http://sourceforge.net/projects/randlm/

- `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<sup>1/`values`</sup>. For a BackoffRandLM a binning quantisation algorithm is used. The size of the codebook is set as 2<sup>`values`</sup>. A reasonable setting in both cases is `-values 8`.
- `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.

Building Randomised LMs from 100 Billion Words using Hadoop

At some point you will discover that you 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 Hadoop (Map-Reduce). The RandLM release has Hadoop scripts which take raw text files and create ngram-counts. We have built randomised LMs this way using more than 110 billion tokens.

The procedure for using Hadoop is as follows:

- You first load raw and possibly tokenised text files onto the Hadoop Distributed File System (DFS). This will probably involve commands such as:

```
Hadoop dfs -put myFile data/
```

- You then create ngram-counts using Hadoop (here a 5-gram):

```
perl hadoop-lm-count.prl data data-counts 5 data-counting
```

- You then upload the counts to the Unix filesystem:

```
perl hadoopRead.prl data-counts | gzip - > /unix/path/to/counts.gz
```

- These counts can then be passed to RandLM:

```
./buildlm -estimator batch  -smoothing WittenBell  -order 5 -values 12 -struct LogFreqBloomFilter
-tmp-dir /disk5/miles -output-prefix giga3.rlm -output-dir /disk5/miles -falsepos 12
-keep-tmp-files -sorted-by-ngram -input-type counts -input-path /disk5/miles/counts.gz
```

Querying Randomised Language Models

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 (subject to randomisation errors).

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`.

- Finally, you then tell randLM to use these pre-computed counts.

4.13.6 KenLM

KenLM is a language model that is simultaneously fast and low memory. The probabilities returned are the same as SRI, up to floating point rounding. It is maintained by Ken Heafield, who provides additional information on his website[15], such as benchmarks[16] comparing speed and memory use against the other language model implementations. KenLM is distributed with Moses and compiled by default. KenLM is fully threadsafe for use with multithreaded Moses.

It can read arpa files directly:

```
8 <factor> <size> filename.arpa
```

but the binary format loads much faster and provides more flexibility. The <size> field is ignored. By contrast, SRI silently returns incorrect probabilities if you get it wrong (Kneser-Ney smoothed probabilties for lower-order n-grams are conditioned on backing off).

Binary file

Using the binary format significantly reduces loading time. It also exposes more configuration options. The kenlm/build_binary program converts ARPA files to binary files:

```
kenlm/build_binary filename.arpa filename.binary
```

This will build a binary file that can be used in place of the ARPA file. Note that, unlike IRST, the file extension does not matter; the binary format is recognized using magic bytes. You can also specify the data structure to use:

[15] http://kheafield.com/code/kenlm/

[16] http://kheafield.com/code/kenlm/benchmark/

```
kenlm/build_binary trie filename.arpa filename.binary
```

where valid values are probing, sorted, and trie. The default is probing. Generally, I recommend using probing if you have the memory and trie if you do not. See benchmarks for details. To determine the amount of RAM each data structure will take, provide only the arpa file:

```
kenlm/build_binary filename.arpa
```

Bear in mind that this includes only language model size, not the phrase table or decoder state.

Building the trie entails an on-disk sort. You can optimize this by passing -m to specify sorting memory in MB (but final model building will still use the amount of memory needed to store the model). The -t option lets you customize where to place temporary files (the default is based on the output file name):

```
kenlm/build_binary -t /tmp/trie -m 1024 trie filename.arpa filename.binary
```

Full or lazy loading

KenLM supports lazy loading via mmap. This allows you to further reduce memory usage, especially with trie which has good memory locality. In Moses, this is controlled by the language model number in moses.ini. Using language model number 8 will load the full model into memory (`MAP_POPULATE` on Linux and `read()` on other OSes). Language model number 9 will lazily load the model using mmap. I recommend fully loading if you have the RAM for it; it actually takes less time to load the full model and use it because the disk does not have to seek during decoding. Lazy loading works best with local disk and is not recommended for networked filesystems.

Probing

Probing is the fastest and default data structure. Unigram lookups happen by array index. Bigrams and longer n-grams are hashed to 64-bit integers which have very low probability of collision, even with the birthday attack. This 64-bit hash is the key to a probing hash table where values are probability and backoff.

A linear probing hash table is an array consisting of blanks (zeros) and entries with non-zero keys. Lookup proceeds by hashing the key modulo the array size, starting at this point in the array, and scanning forward until the entry or a blank is found. The ratio of array size to number of entries is controlled by the probing multiplier parameter p. This is a time-space tradeoff: space is linear in p and time is O(p/(p-1)). The value of p can be set at binary building time e.g.

```
kenlm/build_binary -p 1.2 probing filename.arpa filename.binary
```

sets a value of 1.2. The default value is 1.5 meaning that one third of the array is blanks.

Trie

The trie data structure uses less memory than all other options (except RandLM with stupid backoff), has the best memory locality, and is still faster than any other toolkit. However, it does take longer to build. It works in much the same way as SRI and IRST's inverted option. Like probing, unigram lookup is an array index. Records in the trie have a word index, probability, backoff, and pointer. All of the records for n-grams of the same order are stored consecutively in memory. An n-gram's pointer is actually the index into the (n+1)-gram array where block of (n+1)-grams with one more word of history starts. The end of this block is found by reading the next entry's pointer. Records within the block are sorted by word index. Because the vocabulary ids are randomly permuted, a uniform key distribution applies. Interpolation search within each block finds the word index and its correspoding probability, backoff, and pointer. The trie is compacted by using the minimum number of bits to store each integer. Probability is always non-positive, so the sign bit is also removed.

Since the trie stores many vocabulary ids and uses the minimum number of bits to do so, vocabulary filtering is highly effective for reducing overall model size even if less n-grams of higher order are removed.

Quantization

The trie supports quantization to any number of bits from 1 to 25. To quantize to 8 bits, use -q 8. If you want to separately control probability and backoff quantization, use -q for probability and -b for backoff.

Array compression (aka Chop)

The trie pointers comprise a sorted array. These can be compressed using a technique from Raj and Whittaker by chopping off bits and storing offsets instead. The -a option acts as an upper bound on the number of bits to chop; it will never chop more bits than minimizes memory use. Since this is a time-space tradeoff (time is linear in the number of bits chopped), you can set the upper bound number of bits to chop using -a. To minimize memory, use -a 64. To save time, specify a lower limit e.g. -a 10.

Vocabulary lookup

The original strings are kept at the end of the binary file and passed to Moses at load time to obtain or generate Moses IDs. This is why lazy binary loading still takes a few seconds. KenLM stores a vector mapping from Moses ID to KenLM ID. The cost of this vector and Moses-side vocabulary word storage are not included in the memory use reported by build_binary. However, benchmarks[17] report the entire cost of running Moses.

Subsection last modified on November 07, 2011, at 11:07 AM

[17] http://kheafield.com/code/kenlm/benchmark/

4.14 Tuning

4.14.1 Overview

During decoding, Moses scores translation hypotheses using a linear model. In the traditional approach, the features of the model are the probabilities from the language models, phrase/rule tables, and reordering models, plus word, phrase and rule counts. Recent versions of Moses support the augmentation of these **core features** with **sparse features** (page **??**), which may be much more numerous.

Tuning refers to the process of finding the optimal weights for this linear model, where optimal weights are those which maximise translation performance on a small set of parallel sentences (the **tuning set**). Translation performance is usually measured with Bleu, but the tuning algorithms all support (at least in principle) the use of other performance measures. The interest in sparse features has led to the development of new tuning algorithms, and Moses contains implementations of some of these.

There are essential two types of tuning algorithms used in statistical MT: **batch** and **online**. The algorithms are listed in the following sections.

4.14.2 Batch tuning algorithms

Here the whole tuning set is decoded, usually generating an n-best list or a lattice, then the model weights are updated based on this decoder output. The tuning set is then re-decoded with the new weights, the optimisation repeated, and this iterative process continues until some convergence criterion is satisfied. All the batch algorithms in Moses are controlled by the inaccurately named `mert-moses.pl`, which runs the 'outer loop' (i.e. the repeated decodes). Running this script with no arguments displays usage information.

MERT

(Och 2003)[18], (Bertoldi et al, 2009)[19]

This line-search based method is probably still the most widely used tuning algorithm, and the default option in Moses. It does not support the use of more than about 20-30 features, so for sparse features you should use one of the other algorithms.

PRO

(Hopkins and May, 2011)[20]

This method works by learning a weight set that ranks translation hypotheses in the same order as the metric (e.g. Bleu). Passing the argument `-pairwise-ranked` to `mert-moses.pl` enables PRO.

Batch MIRA

(Cherry and Foster, 2012)[21]

[18] http://aclweb.org/anthology-new/P/P03/P03-1021.pdf

[19] http://homepages.inf.ed.ac.uk/bhaddow/prague-mert.pdf

[20] http://www.aclweb.org/anthology/D11-1125.pdf

[21] http://aclweb.org/anthology-new/N/N12/N12-1047.pdf

Also known as k-best MIRA, this is a version of MIRA (a margin-based classification algorithm) which works within a batch tuning framework. To use batch MIRA, you need to pass the `-batch-mira` argument to `mert-moses.pl`.

4.14.3 Online tuning algorithms

These methods requires much tighter integration with the decoder. Each sentence in the tuning set is decoded in turn, and based on the results of the decode the weights are updated before decoding the next sentence. The algorithm may iterate through the tuning set multiple times.

MIRA

(Chiang, 2012)[22], (Hasler et al, 2011)[23]

This tuning algorithm was inspired by the passive-aggressive algorithms of Koby Crammer, and their application to structured prediction by Ryan MacDonald. After decoding each sentence, MIRA updates the weights only if the metric ranks the output sentence with respect to a (pseudo-)reference translation differently from the model.

Subsection last modified on October 17, 2012, at 11:35 AM

[22] http://www.jmlr.org/papers/volume13/chiang12a/chiang12a.pdf

[23] http://ufal.mff.cuni.cz/pbml/96/art-hasler-haddow-koehn.pdf

5

Background

5.1 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).

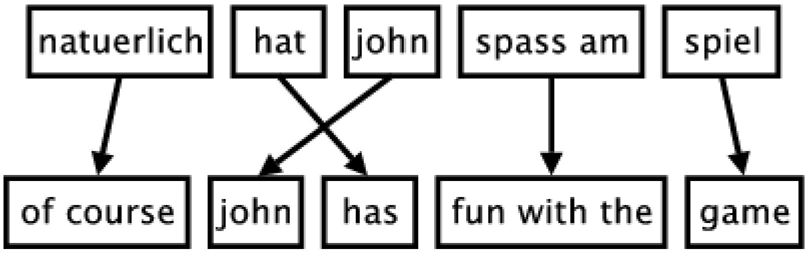
Of course, there are other ways to do machine translation. Most commercial systems use transfer rules and a rich translation lexicon. Until recently, machine translation research has been focused on knowledge based systems that use a interlingua representation as an intermediate step between input and 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.

5.1.1 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

$$\text{argmax}_{\mathbf{e}} p(\mathbf{e}|\mathbf{f}) = \text{argmax}_{\mathbf{e}} p(\mathbf{f}|\mathbf{e})\ p(\mathbf{e})$$

This allows for a language model p(**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 $\mathrm{d}(\mathrm{start}_{\mathrm{i}},\mathrm{end}_{\mathrm{i-1}})$, where $\mathrm{start}_{\mathrm{i}}$ denotes the start position of the foreign phrase that was translated into the *i*th English phrase, and $\mathrm{end}_{\mathrm{i-1}}$ denotes the end position of the foreign phrase that was translated into the *(i-1)*th English phrase.

We use a simple distortion model $\mathrm{d}(\mathrm{start}_{\mathrm{i}},\mathrm{end}_{\mathrm{i-1}}) = \alpha^{|\mathrm{start}_{\mathrm{i}}\text{-}\mathrm{end}_{\mathrm{i-1}}\text{-}1|}$ 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 $\mathbf{e}_{\mathrm{best}}$ given a foreign input sentence **f** according to our model is

$$\mathbf{e}_{\mathrm{best}} = \text{argmax}_{\mathbf{e}} p(\mathbf{e}|\mathbf{f}) = \text{argmax}_{\mathbf{e}} p(\mathbf{f}|\mathbf{e})\ p_{\mathrm{LM}}(\mathbf{e})\ \omega^{\mathrm{length}(\mathbf{e})}$$

where p(**f**|**e**) is decomposed into

$$\mathrm{p}(\bar{f}_1{}^{\mathrm{I}}|\bar{e}_1{}^{\mathrm{I}}) = \Phi_{\mathrm{i=1}}{}^{\mathrm{I}}\ \phi(\bar{f}_i|\bar{e}_i)\ \mathrm{d}(\mathrm{start}_{\mathrm{i}},\mathrm{end}_{\mathrm{i-1}})$$

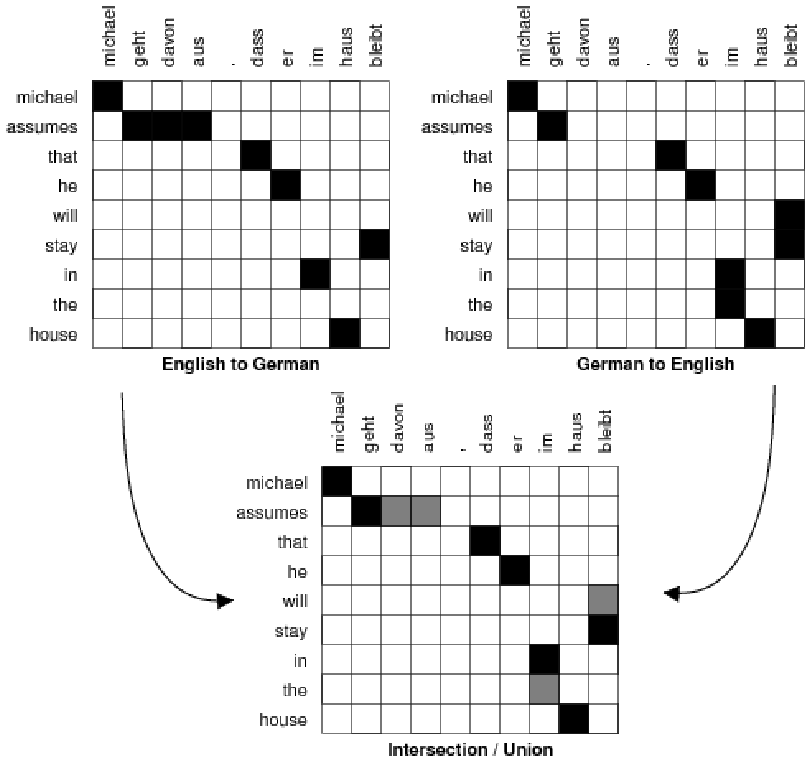
5.1.2 Word Alignment

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.

5.1.3 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

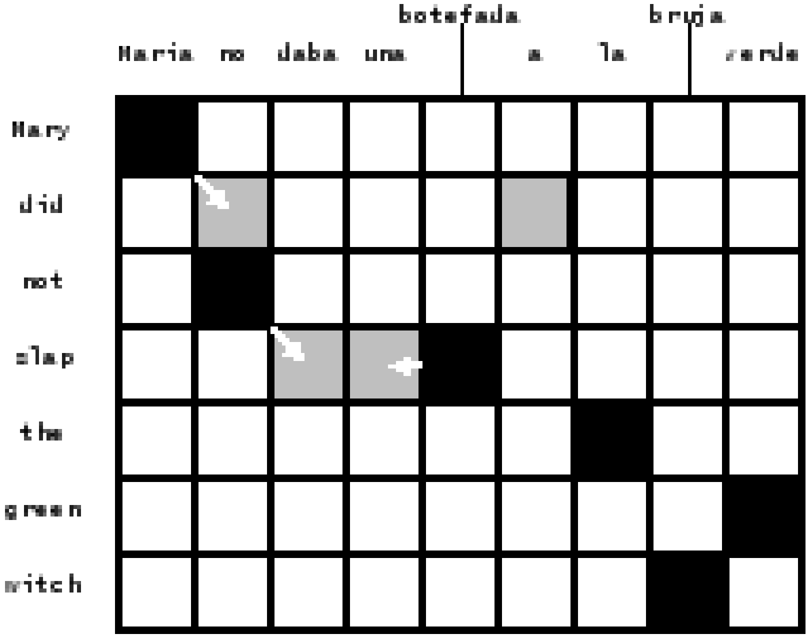
First, the exception: Marcu and Wong (EMNLP, 2002) proposed to establish phrase correspondences directly in a parallel corpus. To learn such correspondences, they introduced a phrase-based joint probability model that simultaneously generates both the source and target sentences in a parallel corpus.

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.

5.1.4 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.

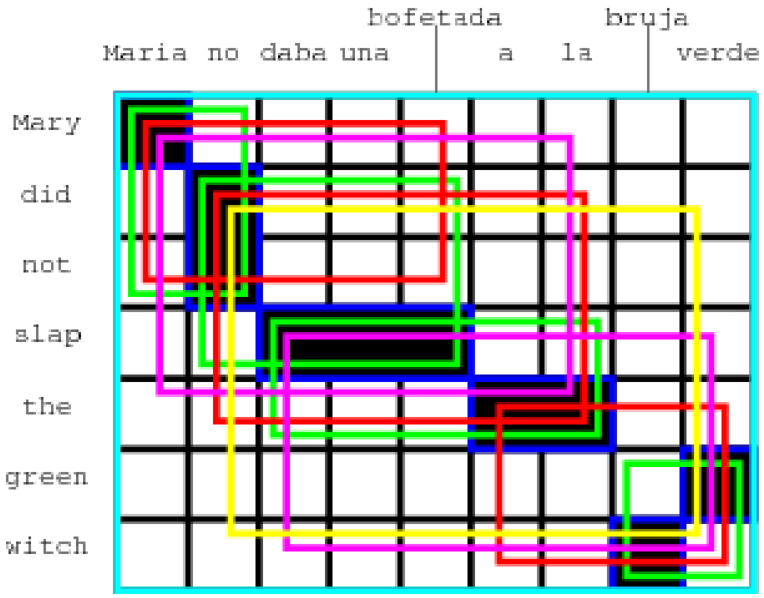
First, we expand to only directly adjacent alignment points. We check for potential points starting from the top right corner of the alignment matrix, checking for alignment points for the first English word, then continue with alignment points for the second English word, and so on.

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.

5.2 Decoder

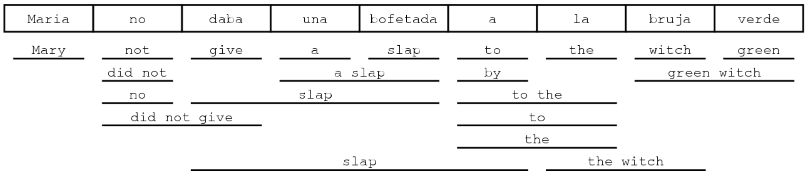
This section describes the Moses decoder from a more theoretical perspective. The decoder was originally developed for the phrase model proposed by Marcu and Wong. At that time, only a greedy hill-climbing decoder was available, which was unsufficent for our work on noun phrase translation (Koehn, PhD, 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.

5.2.1 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

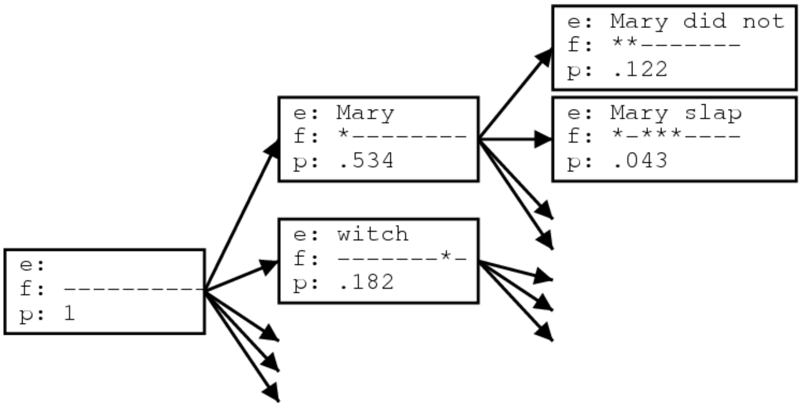
- first foreign word covered
- last foreign word covered
- English phrase translation
- phrase translation probability

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.

5.2.2 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.

5.2.3 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.

5.2.4 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 $N \sim 2^{n_f} |V_e|^2 n_f$ 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.

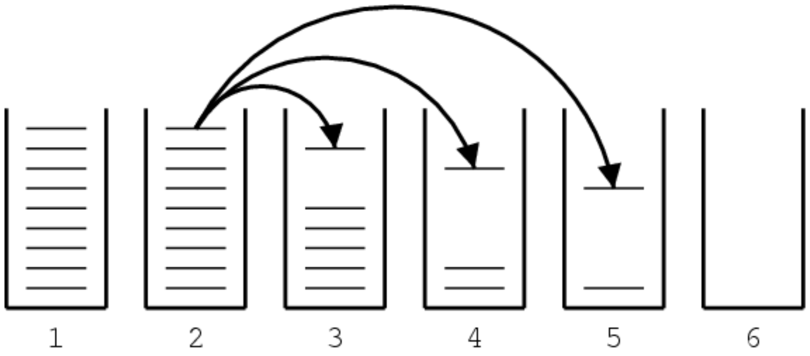
Given the cost so far and the future cost estimation, we can prune out hypotheses that fall outside the beam. The beam size can be defined by threshold and histogram pruning. A relative threshold cuts out a hypothesis with a probability less than a factor α of the best hypotheses (e.g., $\alpha = 0.001$). 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.



5.2.5 Future Cost Estimation

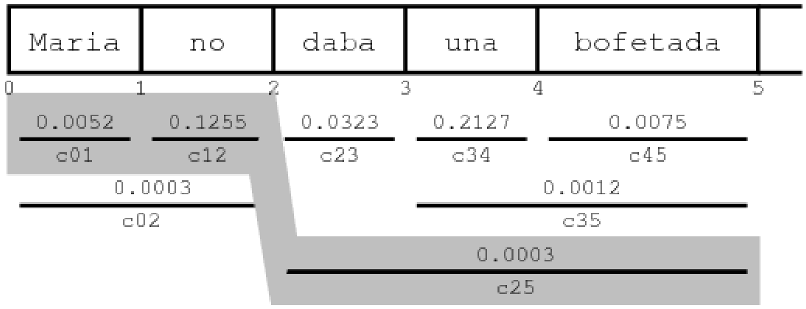
Recall that for excluding hypotheses from the beam we do not only have to consider the cost so far, but also an estimate of the future cost. While it is possible to calculate the cheapest possible future cost for each hypothesis, this is computationally so expensive that it would defeat the purpose of 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.

5.2.6 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).

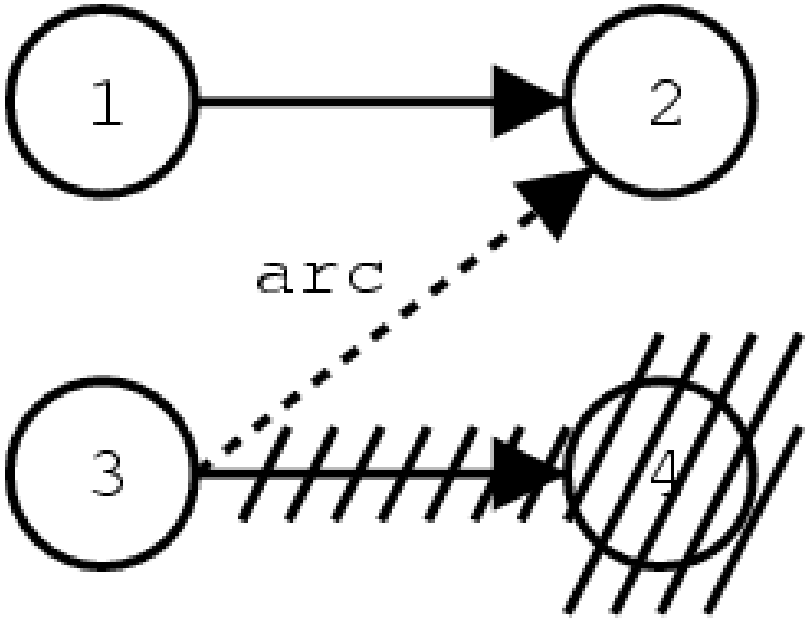
But since we are now also interested in the second best translation, we cannot simply discard information about that hypothesis. If we would do this, the search graph would only contain one path for each hypothesis in the last hypothesis stack (which contains hypotheses that cover all foreign words).

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.

Subsection last modified on April 26, 2012, at 08:07 PM

5.3 Factored Translation Models

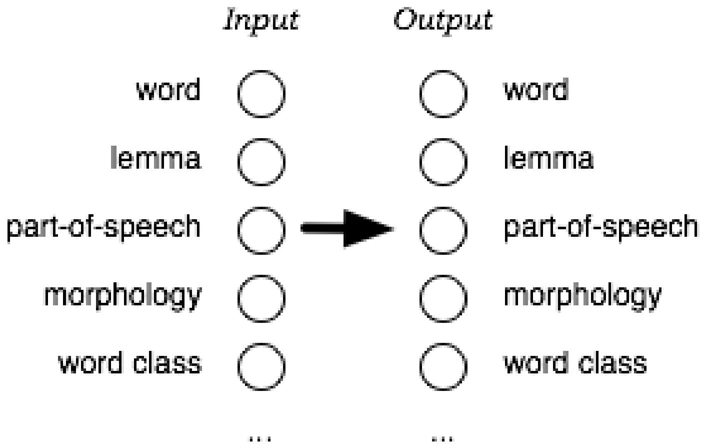
The current state-of-the-art approach to statistical machine translation, so-called phrase-based models, are limited to the mapping of small text chunks (phrases) without any explicit use of linguistic information, may it be morphological, syntactic, or semantic. Such additional information has been 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/



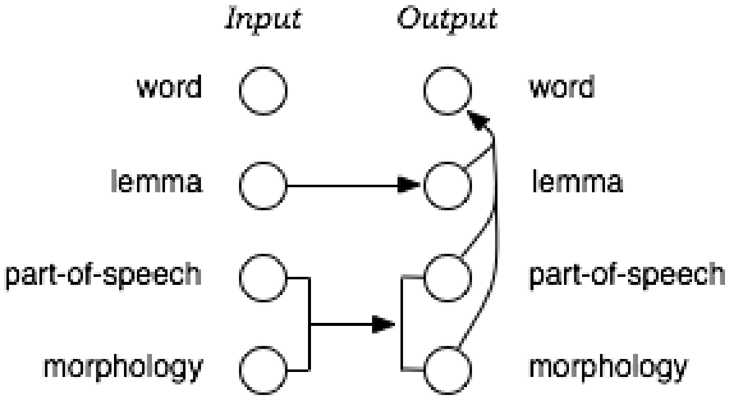
5.3.1 Motivating Example: Morphology

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.

5.3.2 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, ...* }

5.3.3 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 **e** given an input sentence **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.

Subsection last modified on December 22, 2008, at 06:20 AM

5.4 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.

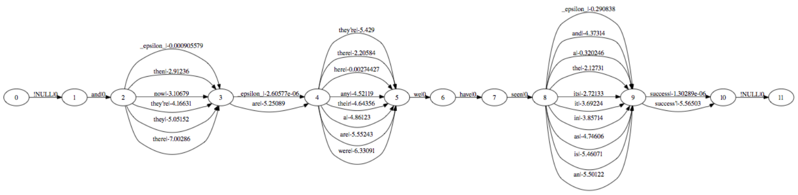
5.4.1 Confusion Networks

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_`).

5.4.2 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
```

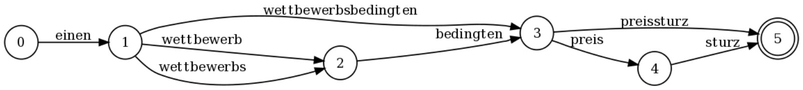
Notice that if you project the above CN on a single factor, repetitions of factors must be merged and the respective probs summed up. The corresponding word-projected CN is the one of the first example, while the part-of-speech projected CN is:

```
N 0.5 PREP 0.4 _eps_ 0.1
DET 0.1 PREP 0.8 _eps_ 0.1
N 0.8 _eps_ 0.2
```

Subsection last modified on February 08, 2009, at 09:51 PM

5.5 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).

5.5.1 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, but see the comments on `-max-phrase-length` below). 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),),)
```

5.5.2 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.

In word lattices, the phrase length limit imposed by the `-max-phrase-length` parameter (default: 20) limits the difference between the indices of the start and the end node of a phrase. If your lattice contains long jumps, you may need to increase `-max-phrase-length` and/or renumber the nodes to make the jumps smaller.

5.5.3 Verifying PLF files with `checkplf`

The command `moses-cmd/src/checkplf` reads a PLF (lattice format) input file and verifies the format as well as producing statistics.

Here’s an example running the application on buggy input:

```
./checkplf < tanaka.plf
Reading PLF from STDIN...
Line 1: edge goes beyond goal node at column position 8, edge label = '&#65332;&#65313;&#65326;&#65313;&#65323;&#65313;'
Goal node expected at position 12, but edge references a node at position 13
```

Here's an example running the application on good input:

```
christopher-dyers-macbook:src redpony$ ./checkplf < ok.plf
Reading PLF from STDIN...
PLF format appears to be correct.
STATISTICS:
Number of lattices: 1
Total number of nodes: 7
Total number of edges: 9
Average density: 1.28571 edges/node
Total number of paths: 4
Average number of paths: 4
```

5.5.4 Citation

If you use Moses's lattice translation in your research, please cite the following paper:

Chris Dyer, Smaranda Muresan, and Philip Resnik. Generalizing Word Lattice Translation[5]. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, July 2008.

Subsection last modified on August 16, 2010, at 01:23 AM

5.6 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.

[5] http://aclweb.org/anthology-new/P/P08/P08-1115.pdf

- 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
- Kenneth Heafield: **KenLM: Faster and Smaller Language Model Queries**, Proceedings of the Sixth Workshop on Statistical Machine Translation (WMT), 2011.
- Marcin Junczys-Dowmunt: **Phrasal Rank-Encoding: Exploiting Phrase Redundancy and Translational Relations for Phrase Table Compression**, Proceedings of the Machine Translation Marathon 2012, The Prague Bulletin of Mathematical Linguistics, vol. 98, pp. 63-74, 2012.

Subsection last modified on September 05, 2012, at 10:00 AM

6

Code Guide

6.1 Code Guide

6.1.1 Github, branching, and merging

If you want to code with Moses, you should create your own repository in 1 of a number of ways.

The preference is that you **4 Fork the repository** if you're doing long-term research. If you fixed a bug, please **5 Commit it yourself**, or **6 Create a pull request**.

- **1 Clone the moses github repository** to your hard disk and work with it:

```
git clone https://github.com/moses-smt/mosesdecoder.git mymoses
cd mymoses
edit files ....
git commit -am "Check in"
```

You don't need a github login or permission to do this. All changes are stored on your own hard disk

- **2 Clone AND branch the repository**:

```
git clone https://github.com/moses-smt/mosesdecoder.git mymoses
cd mymoses
git checkout -b mybranch
edit files ....
git commit -am "Check in"
```

You still don't need a github login or permission to do this.

- **3 Clone and branch AND push to github**:

```
git clone https://github.com/moses-smt/mosesdecoder.git mymoses
cd mymoses
git checkout -b mybranch
edit files ....
git commit -am "Check in"
git push origin mybranch
edit files ....
git commit -am "Check in again"
git push
```

You need a github account. And you have to ask one of the Moses administrators to add you as a committer to the Moses repository.

NB. To delete a LOCAL branch:

```
git branch -D new-branch
```

To delete a branch on the github server:

```
git push origin --delete new-branch
```

- **4 Fork the repository**. You need a github account. You don't need permission from the Moses administrators. Log into github.com on their webiste, and go to the Moses page:

```
https://github.com/moses-smt/mosesdecoder
```

Press the **Fork** button. This creates a new repository only you have write access to. Clone that repository and do whatever you want. Eg.

```
git clone https://github.com/hieuhoang/mosesdecoder.git
cd hieuhoang
edit files ....
git commit -am "Check in again"
git push
```

- **5 Clone and check into master**

```
git clone https://github.com/moses-smt/mosesdecoder.git mymoses
cd mymoses
edit files ....
git commit -am "Check in"
git push
```

You need a github account and write permission to the Moses repository.

- **6 Create pull request**. Fork a repository and read the instructions here:

```
https://help.github.com/articles/using-pull-requests
```

Working with multiple branches

Assuming you've done **4 Fork the repository**, you can merge the latest changes from the main Moses repository with this command:

```
git pull https://github.com/moses-smt/mosesdecoder.git
```

In your own repository, you can create branches and switch between them using

```
git checkout -b new-branch
edit files...
git commit -am "check in"

git checkout master
edit files...
...
```

To get the latest changes from the main Moses repository to your branch, on your fork:

```
git checkout master
git pull https://github.com/moses-smt/mosesdecoder.git
git checkout new-branch
git merge master
```

Regression test

If you've changed any of the C++ code and intend to check into the main Moses repository, please run the **regression test** to make sure you haven't broken anything:

```
git submodule init
git submodule update
./bjam with-irstlm=... --with-srilm=... {\bf -a --with-regtest} >& reg.out &
```

Check the output for any failures:

```
grep FAIL reg.out
```

Contact the administrators

Contact Hieu Hoang or Barry Haddow, or any of the other administrators you might know, if you need help or permission to the github repository.

6.1.2 The code

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 the following directories

- `moses/util` contains some shared utilities, such as code for reading and parsing files, and for hash tables. This was originally part of KenLM.
- `moses/lm` contains KenLM, Moses default language model.
- `moses/src` contains the code for the decoder
- `moses/src/LM` contains the language model wrappers used by the decoder
- Other subdirectories of `moses/src` contain some more specialised parts of the decoder, such as alternative chart decoding algorithms.
- `moses-cmd/src` contains code relevant to the command line version of the phrase-based decoder
- `moses-cmd/src` contains code relevant to the command line version of the chart-based decoder
- `mert` contains the code for the Moses mert implementation, originally described here[2]

In the following, we provide a short walk-through of the decoder.

[1] http://www.statmt.org/moses/html/hierarchy.html

[2] http://homepages.inf.ed.ac.uk/bhaddow/prague-mert.pdf

6.1.3 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
  - `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`

6.1.4 Detailed Guides

- Adding feature functions (page **??**)
- Background on factor, word, and phrase data structure (page **??**)
- Chart decoder (page **??**)
- Style Guide (page **??**)
- Multithreading (page **??**)

Subsection last modified on September 25, 2012, at 10:12 AM

6.2 Coding Style

To ensure maintainability and consistency, please follow the recommendations below when developing Moses.

6.2.1 Formatting

Indentations are 2 spaces. No tab characters allowed in the code. To ensure that your code follows this format run `scripts/other/beautify.perl` in the directory of the source code.

Opening braces are on a separate line, for instance:

```
if (expr) // correct
{
...
}

if (expr) { // wrong
...
}
```

Upper/lowercase: Start all functions and class names with capital letters. Start all variable with small letter. Start all class variable with m_. For instance:

```
void CalcNBest(size_t count, LatticePathList &ret) const;
Sentence m_source;
```

Use **long variable names**, not variables called s, q, or qb7.

Do not use Hungarian notation[3].

6.2.2 Comments

The code will be parsed by Doxygen to create online documentation[4]. To support this, you have to add comments for each class, function, and class variable. More information is available at the Doxygen[5] web site.

Class definitions in the *.h file need to be preceeded by a block starting with /**, for instance:

```
/** The Manager class implements a stack decoding algorithm.
* Hypotheses are organized in stacks. One stack contains all hypothesis that have
* the same number of foreign words translated.  The data structure for hypothesis
* stacks is the class HypothesisStack. The data structure for a hypothesis
* is the class Hypothesis.
[...]
**/
class Manager
```

Class member variable definitions in *.h must be followed by a comment that starts with //!, for instance:

```
size_t m_maxNumFactors;  //! max number of factors on both source and target sides
```

Functions in the *.cpp file need to be preceeded by a block starting with /**, for instance:

[3] http://en.wikipedia.org/wiki/Hungarian_notation

[4] http://www.statmt.org/moses/html/hierarchy.html

[5] http://www.stack.nl/~dimitri/doxygen/docblocks.html

```
/**
* Main decoder loop that translates a sentence by expanding
* hypotheses stack by stack, until the end of the sentence.
*/
void Manager::ProcessSentence()
```

Function parameters are described by `param`, for instance:

```
/** Create translation options that exactly cover a specific input span.
* Called by CreateTranslationOptions() and ProcessUnknownWord()
* \param decodeGraph list of decoding steps
* \param factorCollection input sentence with all factors
* \param startPos first position in input sentence
* \param lastPos last position in input sentence
* \param adhereTableLimit whether phrase & generation table limits are adhered to
*/
void TranslationOptionCollection::CreateTranslationOptionsForRange(
const DecodeGraph &decodeGraph
, size_t startPos
, size_t endPos
, bool adhereTableLimit)
{
```

In addition the definition in the `*.h` must be preceded by a short comment that starts with `//!`. This comment will be displayed in the beginning of the class definition. For instance:

```
//! load all language models as specified in ini file
bool LoadLanguageModels();
```

6.2.3 Data types and methods

- Code for cross-platform compatibility.
- 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.

6.2.4 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.

Subsection last modified on April 26, 2012, at 08:29 PM

6.3 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.

6.3.1 Factors

The class `Factor`[6] 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`[7] 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.

6.3.2 Words

A word is, as we said, a vector of factors. The class `Word`[8] 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.

[6] http://www.statmt.org/moses/html/classMoses_1_1Factor.html

[7] http://www.statmt.org/moses/html/d2/d9f/classMoses_1_1FactorCollection.html

[8] http://www.statmt.org/moses/html/d7/dc4/classMoses_1_1Word.html

6.3.3 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`[9], which is a bit array indicating which factors are valid for a particular word.

6.3.4 Phrases

Since decoding proceeds in the translation of input phrases to output phrases, a lot of operation involve the class `Phrase`[10]. 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:

- The simplest form of input, a sentence as string of words, is implemented in the class `Sentence`[11].
- The class `TargetPhrase`[12] 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.

Subsection last modified on April 26, 2012, at 08:34 PM

6.4 Tree-Based Model Decoding

The chart decoder is a implementation of CKY+ parse/decoding which is able to process arbitrary context free grammars with no limitations on the number of terminals or non-terminals in a rule. During decoding, all contiguous spans over the input spans are filled with hypotheses (partial translations). Rules are stored in a prefix tree, which is processed incrementally, in a way akin to Early parsing. Once rules are looked up, cube pruning is applied to pick off the most likely applicable rules and hypotheses from underlying spans.

[9] http://www.statmt.org/moses/html/db/d41/classMoses_1_1FactorMask.html

[10] http://www.statmt.org/moses/html/df/d4d/classMoses_1_1Phrase.html

[11] http://www.statmt.org/moses/html/d3/dfd/classMoses_1_1Sentence.html

[12] http://www.statmt.org/moses/html/db/d67/classMoses_1_1TargetPhrase.html

6.4.1 Content

- Looping over the Spans (page **??**)
- Consulting Rule Tables (page **??**)
- Looking up Applicable Rules (page **??**)
- DottedRuleColl (in DotChart) (page **??**)
- DottedRule (in DotChart) (page **??**)
- CoveredChartSpan (page **??**)
- ChartTranslationOptionList and ChartTranslationOption (page **??**)
- ChartTranslationOption (page **??**)
- PhraseDictionarySCFG (page **??**)
- TargetPhraseCollection (page **??**)
- PhraseDictionaryNodeSCFG (page **??**)
- Applying the Rules: Cube Pruning (page **??**)
- ChartCell::ProcessSentence (page **??**)
- ChartCell (page **??**)
- RuleCubeQueue (page **??**)
- RuleCube (page **??**)
- Hypotheses and Pruning (page **??**)

6.4.2 Looping over the Spans

ChartManager::ProcessSentence[13]

The main loop of the decoding process fills up the stack bottom up: first looping of the width of the span, and then over the starting position of the span.

```
for (size_t width = 1; width <= size; ++width) {
for (size_t startPos = 0; startPos <= size-width; ++startPos) {
```

For each span, first the applicable rules are created and then the rules are applied.

[13] http://www.statmt.org/moses/html/d9/dde/classMoses_1_1ChartManager.html

```
m_transOptColl.CreateTranslationOptionsForRange(startPos, endPos);
ChartCell &cell = m_hypoStackColl.Get(range);
cell.ProcessSentence(m_transOptColl.GetTranslationOptionList(range)
,m_hypoStackColl);
```

Processing a span concludes with pruning, cleaning, and sorting of the hypotheses that were placed into the span.

```
cell.PruneToSize();
cell.CleanupArcList();
cell.SortHypotheses();
```

Consulting Rule Tables

ChartTranslationOptionCollection::CreateTranslationOptionsForRange[14]

Get existing data (or pointer?) from global rule collection

```
ChartTranslationOptionList &chartRuleColl = GetTranslationOptionList(startPos, endPos);
```

Multiple rule tables may be consulted. In fact, in most setups there will be a main rule table and a rule table for glue rules. So, we need to consult each of them.

```
ChartRuleLookupManager &ruleLookupManager;
ruleLookupManager.GetChartRuleCollection(wordsRange, true, chartRuleColl);
```

6.4.3 Looking up Applicable Rules

ChartRuleLookupManager::GetChartRuleCollection[15]

There are multiple implementations of the rule table. At the time of this writing, there are two: one that is kept entirely in memory and a second one that keeps the data on disk. There is an obvious RAM/speed trade-off here: the in-memory rule table is faster, but for some models there may not be sufficient RAM. Both are implemented very similarly, however.

[14] http://www.statmt.org/moses/html/de/dae/classMoses_1_1ChartTranslationOptionCollection.html

[15] http://www.statmt.org/moses/html/dd/d62/classMoses_1_1ChartRuleLookupManager.html

For a given span, there may be many rules that could apply. Each rule may consume input words directly or build on any number of hypotheses that were generated for sub-spans. There is a combinatorial explosion of sub-spans and input words that could be combined, if there is a rule in the rule table.

The implementation of rule lookup is inspired by Early parsing, which allows for incremental lookup. Consider the English-German rule:

```
PP/NP -> of the JJ_1 problem of NP_2 ; des ADJ_1 Problems NP-GEN_2
```

This is a good time to clarify some terminology:

- PP is the source side parent non-terminal
- NP is the target side parent non-terminal
- JJ and NP are the source side child non-terminals
- of, the, problem and of are the source side child terminals (or words)
- ADJ and NP-GEN are the target side child non-terminals
- des and Problems are the target side child terminals (or words)

Instead of *child*, the term *right hand side*, and correspondingly instead of *parent*, the term *left hand side* is often used.

To check if the rule matches, we have to see if there are indeed English words of, the, problem, and of in the input and the intervening spans have the label JJ and NP. In addition, the rule can only apply if we have hypotheses with the constituent labels ADJ and NP-GEN in the corresponding spans.

We store the rule in a prefix structure with the following nodes:

```
of -> the -> JJ/ADJ -> problem -> of -> NP/NP-GEN -> des ADJ_1 Problems NP-GEN_2
```

The key insight is the following: If there is such an applicable rule for the span under consideration, then a sub-span with the same start position matched part of this prefix path, namely:

```
of -> the -> JJ/ADJ -> problem -> of
```

In bottom-up parsing, the sub-span was explored before the current span. If we saved a pointer to this path, then we now we only have to check if it can be extended by a NP/NP-GEN. Note: It does not matter, if there was actually a translation rule associated with the prefix path, all it matters that it matched the sub-span earlier.

Hence, for each starting position, we maintain a list of matched paths in a list called DottedRuleList with each starting position. Note that a DottedRule is technically not a rule, just a match to the span, which may have translations rules associated with it. During rule lookup for a new span, we try to extend these processed rules by just one non-terminal or by just one terminal.

The code in `ChartRuleLookupManagerMemory::GetChartRuleCollection`[16] implements the lookup that we just described. It is extensively commented, so that is should be understandable without further explanation.

From the outside, each time the function is called for a span, it updates its internal data structures for processed rules and returns applicable rules for the span (with their target side) as a `ChartTranslationOptionList`.

??? why do we loop over processed rules that are generated (for (size_t ind = 0; ind < ExpandableDottedRuleList.size(); ++ind)) - this implies that more than one word / non-terminal could be added, but I do not see how this is possible.

Let us take a closer look at the data structures used in rule lookup in the function `ChartRuleLookupManagerMemory`

DottedRuleColl (in DotChart)

The main data stored in a rule collection (for a given starting position in the chart) are the `DottedRules` that start at a given start position. They are grouped by end position, so the rule collection is a vector of `DottedRuleLists`, each of which is a vector of `DottedRules`.

In addition to that comprehensive double vector of `DottedRules`, the data structure maintains a subset of the rules in a second `DottedRuleList`, called `m_expandableDottedRuleList`. It contains only those processed rules that can be extended, in other words, that have not reached a leaf in the prefix tree.

The core operations of the data structure are `Add`, which adds a processed rule (which ends at a specified end position); `GetExpandableDottedRuleList`, which returns the expandable processed rules; and `Get` which returns the processed rules that end at a specified end position.

DottedRule (in DotChart)

A processed rules has two data structures: a pointer to the prefix tree (of type `PhraseDictionaryNodeSCFG`), and the encoding of the rule and how it matches the chart (in a pointer to a `CoveredChartSpan`)

Note: A pointer to the root of the prefix tree is encoded as a `DottedRule` with `CoveredChartSpan` = `NULL`.

These have to specified when creating a `DottedRule` and can be accessed with the functions `GetLastNode` (pointer to the prefix tree) and `GetLastCoveredChartSpan` (pointer to the matched rule).

Some stuff is going on during instantiation for each word position (in initializer for `ChartRuleLookupManagerMemo`

```
m_dottedRuleColls[i]->Add(0, initDottedRule);
```

CoveredChartSpan

Recall that processed rules are lookup up using a prefix tree, for example:

[16] http://www.statmt.org/moses/html/df/d0b/classMoses_1_1ChartRuleLookupManagerMemory.html

[17] http://www.statmt.org/moses/html/df/d0b/classMoses_1_1ChartRuleLookupManagerMemory.html

```
of -> the -> JJ/ADJ -> problem -> of -> NP/NP-GEN -> des ADJ_1 Problems NP-GEN_2
```

To use the rule, we need to know how each of these nodes in this path matches the chart, in other words: how each node matches a span. This is encoded in a linked list of `CoveredChartSpans`.

Each `CoveredChartSpan` contains the start and end position in the span (store in a `WordsRange`, which is essential a pair of integers with additional utility functions), the source word or source span label that is matched (as a `Word`), and a back-pointer to the previous `CoveredChartSpan` (hence the linked list).

The linked list of `CoveredChartSpans` contains all the necessary information about applying the rule to the source side. Target side information is stored elsewhere. Note that target-side non-terminals are not stored anymore, since they will be contained in the target side information.

The core operations of the data structure are `GetSourceWord`, `GetWordsRange`, and `GetPrevCoveredChartSpan`. There is also the utility function `IsNonTerminal` which checks if the last word in the linked list is a non-terminal.

ChartTranslationOptionList and ChartTranslationOption

A `ChartTranslationOptionList` is a vector of `ChartTranslationOptions`, with additional utility functions.

Rules are added to this in batches, since rules are stored in the prefix tree first by matching the source words and child non-terminals, pointing towards a list of target sides.

The list contains fully fledged out rules with target sides (opposed to the cube pruning algorithm described by Chiang, where rules are groups modulo the target side words). The list also observes rule limits, i.e., the maximum number of rules considered for a span. It sorts the rules by the future score of the target side (weighted translation model costs and language model estimates), and prunes out the worst ones when the limit is exceeded. Internally a score threshold is kept, to not even add rules that would be pruned out anyway.

??? Add has a unused `adhereTableLimit` parameter - is this limit applied elsewhere?

Note that when adding `ChartTranslationOption` ultimately some additional processing has to be done — the computation of the alignment between non-terminals by calling `ChartTranslationOption::Cre` This is done lazily, once the list finalized and pruned.

ChartTranslationOption

A `ChartTranslationOption` contains all the information about a rule and its application to the span. It contains a linked list of `CoveredChartSpan` which details how the rule matches the input side, a `TargetPhrase` with the output, and the span it applies to (in a `WordsRange`).

This information has to be specified at instantiation and can be queried with the functions `GetLastCoveredChartSpan`, `GetTargetPhrase`, and `GetSourceWordsRange`.

Once created, the mapping between the source and target non-terminals is computed by calling `CreateNonTermIndex` and can be queried with `GetCoveredChartSpanTargetOrder`. That information is already stored with the `TargetPhrase`, but it is reformated here for easier querying (at the cost of a higher memory footprint).

??? describe GetTotalScore() const { return m_targetPhrase.GetFutureScore(); }

PhraseDictionarySCFG

This class implements the prefix tree that contains the rules.

TargetPhraseCollection

PhraseDictionaryNodeSCFG

6.4.4 Applying the Rules: Cube Pruning

Above, we described how all the rules that apply to the current span are retrieved. In fact, more than that is done: we also annotate each rule how it applies to the span, especially how the non-terminals match the sub-spans.

Applying a rule now only requires the selection of the hypotheses in the specified sub-spans that match the non-terminals in the rule.

To repeat our example:

```
PP/NP -> of the JJ_1 problem of NP_2 ; des ADJ_1 Problems NP-GEN_2
```

We have already identified which sub-spans the target non-terminals `ADJ` and `NP-GEN` apply to. For each of these, however, we may have multiple choices (note that there will be at least one for each, otherwise we would not consider the rule).

The term *cube* pruning derives from the fact that we explore for each rule a multi-dimensional space, with one dimension for each non-terminal (and, in the original Chiang algorithm, but not here, one dimension for the target phrase). This space is not always a cube, only if there are three dimensions (two non-terminals in the Chiang algorithm), and even then it is not a cube because the dimensions typically have differing lengths. And it is not technically a pruning algorithm (which removes bad examples after the fact), but a greedy search for the best rule applications. But hey, what's in a name?

Given the *cube*, we sort each dimension, so that the most promising rule application is in the top left front corner. Most promising, because for each of the non-terminals, we use the matching hypothesis with the best score (and the target phrase with the best future cost estimate).

Finally recall that there are multiple cubes, one for each applicable rule.

ChartCell::ProcessSentence[18]

The cube pruning algorithm is given two data structures, a `ChartTranslationOptionList` that contains all applicable rules (they are now called `ChartTranslationOption`) and the `ChartCellCollection` that contains the chart as it has been filled in so far. The cube pruning algorithm is housed in the `ChartCell` that corresponds to the span we are now filling.

[18] http://www.statmt.org/moses/html/d0/dd7/classMoses_1_1ChartCell.html

First, we have to build the RuleCubeQueue. Recall, how each applicable rule has a *cube*. Well, we throw them all together into one big cube (not really a regular geometrical shape, since the dimensions differ for each rule application). Be that as it may, the first part of the algorithm loops through the ChartTranslationOptionList and creates a RuleCube and adds it to the cube.

The RuleCubeQueue keeps the RuleCubes sorted, so that we can always pop off the most promising rule application with its most promising underlying sub-span hypotheses. For a specified number of times (staticData.GetCubePruningPopLimit())

- we pop off the most promising RuleCube (by calling ruleCubeQueue.Pop())
- build a new ChartHypothesis and calculate its score (hypo->CalcScore())
- add the hypothesis to the ChartCell
- add add the neighbors of the hypothesis to the RuleCubeQueue (ruleCube->CreateNeighbors(ruleCube

ChartCell

A chart cell contains the hypothesis that were created for a span. These hypothesis are grouped by their target side non-terminal.

RuleCubeQueue

RuleCubeQueue is a priority queue of RuleCubes (candidate rule applications). Initially it contains the top-left-front rule application for each ChartTranslationOption. When these are expanded, their neighbors are added to the RuleCubeQueue.

Note that the same neighbor might be reached in multiple ways. If the rule applications (0,0,0), (1,0,0) and (0,1,0) are popped off, then the latter two point to (1,1,0). This is checked in RuleCubeQueue, which does not allow insertion of duplicates.

RuleCube

This class contains the *cube* for a rule. It contains information about the ChartTranslationOption and the a list of underlying sub-span hypotheses for each non-terminal in the rule. The latter is represented as a vector of ChildEntrys, which are essentially ordered lists of hypotheses with additional utility functions.

When the RuleCube is created from a ChartTranslationOption, the vector of ChildEntrys is assembled from the information in the chart. Also, the estimated score of the top-left-front rule application is computed and stored. Note that this is a estimated score, it does not have the real language model cost.

A RuleCube always points to a particular rule application (i.e., particular sub-span hypotheses) in the cube. If it is picked to create an hypothesis, then its neighbors are added to the RuleCubeQueue — this is implemented in the function CreateNeigbors. Consequently, for a particular ChartTranslationOption, there may be multiple RuleCubes in the RuleCubeQueue.

6.4.5 Hypotheses and Pruning

New hypotheses are build in the ChartCell::ProcessSentence[19] function from a RuleCube.

[19] http://www.statmt.org/moses/html/d0/dd7/classMoses_1_1ChartCell.html

```
ChartHypothesis *hypo = new ChartHypothesis(*ruleCube, m_manager);
hypo->CalcScore();
AddHypothesis(hypo);
```

A `ChartHypothesis` contains various type of information about a entry in the chart, i.e., a translation of the covered span.

- Book-keeping
  - `size_t m_id` hypothesis ID
  - `Manager& m_manager` reference to manager
  - `WordsRange m_currSourceWordsRange` covered span
  - `ChartTranslationOption &m_transOpt` rule that created it
  - `vector<size_t> &m_coveredChartSpanTargetOrder` covered sub-spans
- Scores
  - `ScoreComponentCollection m_scoreBreakdown` all scores
  - `ScoreComponentCollection m_lmNGram` language model scores
  - `ScoreComponentCollection m_lmPrefix` estimated language model scores for prefix
  - `float m_totalScore` total weighted score
- Information relevant for recombination and later use
  - `Phrase m_contextPrefix` first words (not yet fully LM-scored)
  - `Phrase m_contextSuffix` last words (affect later attached words)
  - `size_t m_numTargetTerminals` length of phrase (number of words)
- Back-tracking
  - `ChartHypothesis *m_winningHypo` points to superior hypothesis if recombined away
  - `ArcList *m_arcList` all arcs that end at the same trellis point as this hypothesis
  - `vector<const ChartHypothesis*> m_prevHypos` underlying hypotheses

When a hypothesis is created, the *book-keeping* and *information relevant for recombination and later use* is set.

Scores are computed by the function `CalcScore`, by adding up the scores from the underlying hypothesis, the rule application, and language model scoring of the resulting phrase. Language model scoring (in function `CalcLMScore`) is a bit complex, since we do not want to re-compute any of the language model scores that we already computed for the underlying hypotheses. See the documented code for details.

Hypothesis recombination is handled by `ChartHypothesisCollection`. The function `Add` is called to check if the hypothesis is recombinable with anything already in the collection (the class `ChartHypothesisRecombinationOrderer` handles state matching and calls `ChartHypothesis::LMContextC`
`AddHypothesis` calls `Add`, and handles the recombination by potentially replacing the existing hypothesis and setting arcs (`ChartHypothesis::AddArc`).

Subsection last modified on April 26, 2012, at 08:40 PM

6.5 Multi-Threading

Modern server machines have multiple CPUs, each with multiple cores. Utilizing all these cores requires either running multiple processes on the same machine or writing programs that use multiple threads.

Since many aspects of a machine translation system (training, tuning, using) lend themselves very easily to parallel processing, Moses increasingly uses multi-threading in its components. At this point, the following components allow for parallel execution when the switch "–threads NUM" is added with an appropriate maximum number of threads executed at the same time:

- the decoder binary `moses`
- the minimum error rate training tuner `mert`
- the hierarchical rule extractor `extract-rules`

Multi-threading in Moses is based on the C++ Boost libraries, and two Moses helper libraries that make the type of multi-threading that is typical for Moses more convenient: `ThreadPool` and `OutputCollector`.

We will explain the implementation of multi-threaded processing on hand of a simple example.

6.5.1 Tasks

The part of the program that is to be run in parallel threads is called a **task**, and it needs to be placed into a class of its own.

```
class ExampleTask : public Moses::Task
{
public:
ExampleTask() {}
~ExampleTask() {}

void Run() {
std::cout << "Hello World." << endl;
}
}
```

Such a class now allows to be instantiated and run:

```
ExampleTask *task = new ExampleTask()
new->Run();
delete(new);
```

This will print "`Hello World.`", and is otherwise not very exciting.

Let's make the task a bit more interesting. Our new tasks waits for a random amount of time and then prints out a message:

```
ExampleTask(string message):m_message(message) {}

void Run() {
// length of pause
int r = rand()%10;

// pause
int j = 0;
for(int i=0; i<1e8*r; i++) { j+=i; }

// write message (and length of pause)
std::cout << m_message << " (" << r << ")" << endl;
}
```

We can now create multiple instances of this task, and execute each:

```
// set up tasks
srand(time(NULL));
ExampleTask *task0 = new ExampleTask("zero");
ExampleTask *task1 = new ExampleTask("one");
ExampleTask *task2 = new ExampleTask("two");

// serial execution
task0->Run();
task1->Run();
task2->Run();
```

This will print out three lines (the random numbers in parenthesis will vary):

```
zero (2)
one (4)
two (5)
```

Okay, where is the multi-threading? Here it comes.

6.5.2 ThreadPool

Instead of simply running one of the tasks after the other, we assign them to a thread pool. Once assigned, they are spawned off to a thread and will be executed in parallel to the running main process.

```
// set up thread pool
int thread_count = 10;
Moses::ThreadPool pool(thread_count);

// submit tasks
pool.Submit(task0);
pool.Submit(task1);
pool.Submit(task2);

// wait for all threads to finish
pool.Stop(true);
```

That's all too easy to be true, right? Yes, it is.

Since the three threads are running in parallel, there is no telling when they print out their message. Not only could the lines be printed in a different order then the tasks were scheduled, the threads may even write all over each other.

This is the catch with multi-threading: any interaction with non-local data structures must be handled very carefully. Ideally, threads only change local data (defined in the class), and once they are done (after `pool.Stop(true)`), results can be read out. This is in fact what happens in multi-threaded `mert`[20].

In our case, as in the decoder, we want to output text line by line (the decoder outputs translation, and possibly additional information such as n-best lists).

6.5.3 OutputCollector

The Moses code offers the class `OutputCollector` to buffer up the output until it is safe to print out. In the simplest case, it prints to `STDOUT`, but it can also write to a file, and indeed it offers both regular output (default `STDOUT`) and debugging output (default `STDERR`), which both can be redirected to different files.

```
Moses::OutputCollector* outputCollector = new Moses::OutputCollector();
```

A task can then send its output to the output collector with the function `Write`, for example:

[20] http://www.statmt.org/moses/html/d6/d7d/mert_8cpp_source.html

```
m_collector->Write(id, "Hello World!");
```

The id is the sequential number of the sentence, starting at 0. This helps the output collector to keep track of what can be written out and what needs to be buffered. The output collector will not write output for sentence 1, if it has not yet received output for sentence 0.

6.5.4 Not Deleting Threads after Execution

By default, the Task objects are deleted after execution. However, you may want to keep the objects around. This happens for instance in mert, where each Task finds an optimized weight setting, which is to be processed afterwards. In this case, you have to add the following lines to your Task definition:

```
virtual bool DeleteAfterExecution() {
return false;
}
```

6.5.5 Limit the Size of the Thread Queue

By default, when a thread is submitted to the ThreadPool by calling its Submit() function, it is added to an internal queue, and the main process immediately resumes. That means, if a million threads are scheduled, the thread queue is filled with a million instances of the Task, which may consume a lot of memory.

If you want to restrict the number of threads in the queue, you can call, say, pool.SetQueueLimit(1000) to limit it to 1000 queued Task instances. When the queue is full, Submit() blocks.

6.5.6 Example

Below now the complete example.

Note:

- The task class has now two more class variables which are set upon instantiation: the sequence id m_id (a sequential number starting at 0), and a pointer to the output collector m_collector.
- Always implement a fallback to non-threaded compilation (#ifdef WITH_THREADS .. #else .. #endif)
- Output is placed into a file named output-file.txt (lines 43-45) instead of STDOUT.

```
#include <iostream>
#include <fstream>
#include <ostream>
#include <cstdlib>
#include <sstream>
#include "ThreadPool.h"
#include "OutputCollector.h"

using namespace std;

class ExampleTask : public Moses::Task
{
private:
  unsigned int m_id;
  string m_message;
  Moses::OutputCollector* m_collector;
public:
  ExampleTask(unsigned int id, string message, Moses::OutputCollector* collector):
    m_id(id),
    m_message(message),
    m_collector(collector) {}

  ~ExampleTask() {}

  void Run() {
    // length of pause
    int r = rand()%10;

    // pause
    int j = 0;
    for(int i=0; i<1e8*r; i++) { j+=i; }

    // write message (and length of pause)
    ostringstream out;
    out << m_message << " (" << r << ")" << endl;
    m_collector->Write(m_id, out.str());
  }
};

int main ()
{
  // output into file
  string outfile = "output-file.txt";
  std::ofstream *outputStream = new ofstream(outfile.c_str());
  Moses::OutputCollector* outputCollector = new Moses::OutputCollector(outputStream);

  // set up tasks
  srand(time(NULL));
  ExampleTask *task0 = new ExampleTask(0,"zero",outputCollector);
  ExampleTask *task1 = new ExampleTask(1,"one",outputCollector);
  ExampleTask *task2 = new ExampleTask(2,"two",outputCollector);

#ifdef WITH_THREADS
  // set up thread pool
  int thread_count = 10;
  Moses::ThreadPool pool(thread_count);

  // submit tasks
  pool.Submit(task0);
  pool.Submit(task1);
  pool.Submit(task2);

  // wait for all threads to finish
  pool.Stop(true);
#else
  // fallback: serial execution
  task0->Run();
  task1->Run();
  task2->Run();
#endif
}
```

To compile this, you need to copy `ThreadPool.h`, `ThreadPool.cpp`, and `OutputCollector.h` into you code directory or add paths so that they point to the `moses/src` directory and compile as follows:

```
g++ -c ThreadPool.cpp -DWITH_THREADS -DBOOST_HAS_PTHREADS
g++ -c test.cpp -DWITH_THREADS -DBOOST_HAS_PTHREADS
g++ -o test test.o ThreadPool.o -pthread -lboost_thread-mt
```

Make sure that the Boost libraries are in you compile paths.

When you run this example you will notice that, whatever the lengths of the pauses, the output always appears in the correct order (i.e. zero, one, two).

Subsection last modified on April 17, 2012, at 11:49 PM

6.6 Adding Feature Functions

(note: thanks to Jason Katz-Brown for a earlier version)

History:
April 13th, 2012: Checked and revised for latest version (Barry Haddow)

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

$$p(\mathbf{e}|\mathbf{f}) = \prod_i h_i(\mathbf{e}, \mathbf{f})^{\lambda_i} \tag{6.1}$$

where $\mathrm{h_i}$ are the feature functions and λ_i the corresponding weights.

Note that the decoder internally uses logs, so in fact what is computed is

$$\log p(\mathbf{e}|\mathbf{f}) = \sum_i \log(h_i(\mathbf{e},\mathbf{f}))\ \lambda_i \tag{6.2}$$

The tuning (page **??**) stage of the decoder is used to set the weights.

The following components are typically used:

- phrase translation model (5 features, described here (page **??**))
- language model (1 feature)
- distance-based reordering model (1 feature)
- word penalty (1 feature)
- lexicalized reordering model (6 features, described here (page **??**))

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`.

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.

6.6.1 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(unsigned) const;` This function returns the name of the feature (by convention without spaces).
- `std::string GetScoreProducerWeightShortName(unsigned) 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

- `FFState* Evaluate( const Hypothesis&, const FFState*, ScoreComponentCollection* ) const;` Like its stateless counterpart, this function calculates the feature values for the new hypothesis, but in addition it creates and returns a new state object.
- `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(unsigned) const
{
return "XXXFeature";
}

std::string XXX::GetScoreProducerWeightShortName(unsigned) 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;
}

FFState* XXX::EvaluateChart( const ChartHypothesis& cur_hypo,
int featureID,
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.

6.6.2 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.

6.6.3 Integrating the Feature

StaticData.h must include a declaration of the class you are using.

```
class XXX;
```

Also 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
```

In StaticData.cpp, some code needs to be added to process the weight parameter and to create an instance of the score producer.

On top of the file, you need to add a line

```
#include "XXX.h"
```

The feature instance(s) m_XXX have to be initialized to NULL in StaticData::StaticData and deleted in StaticData::~StaticData:

```
delete m_XXX; // if only one instance is possible
RemoveAllInColl(m_XXX); // if multiple instances are possible
```

The function StaticData::LoadData is called upon start-up of the decoder and carries out all the necessary initializations. You 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
}
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.

You will also have to make sure that the TranslationSystem object has a handle to your feature, otherwise it won't get called. Do this by inserting some code in StaticData.cpp, below the comment which states "Add any other features here.". Something like the following should do the trick

```
if (m_XXX != NULL) // check if feature is used
{
m_translationSystems.find(config[0])->second.AddFeatureFunction(m_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.

A simple way to check if the feature is being used is to configure Moses with a higher verbosity, like so:

```
./bin/moses -f moses.ini -v 2 < input > output >& log
```

In the log, look for the place where the score components are listed. It is near the beginning of the log file. Below is an example.

```
The score component vector looks like this:
Distortion
WordPenalty
!UnknownWordPenalty
LexicalReordering_wbe-msd-bidirectional-fe-allff_1
LexicalReordering_wbe-msd-bidirectional-fe-allff_2
LexicalReordering_wbe-msd-bidirectional-fe-allff_3
LexicalReordering_wbe-msd-bidirectional-fe-allff_4
LexicalReordering_wbe-msd-bidirectional-fe-allff_5
LexicalReordering_wbe-msd-bidirectional-fe-allff_6
LM_3gram
PhraseModel_1
PhraseModel_2
PhraseModel_3
PhraseModel_4
PhraseModel_5
XXXFeature
```

6.6.4 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 `TranslationSystem::InitializeBeforeSentenceProcessing(InputType const& in)` in `TranslationSystem.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 you want your feature to be thread-safe (you do, don't you?) then be careful what you do in this function as it may be accessed by multiple threads simultaneously. In particular, it isn't a good idea to update member variables of the feature function as these are shared across threads. If you want to store the results of pre-computation then you could use thread local storage for this, since each sentence is processed in its own thread.

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.

6.6.5 Tuning

Training the weight(s) of your new feature with `mert-moses.pl` is straightforward. The only thing you need to do is to update the `@ABBR_FULL_MAP` variable in the script with the mapping from your feature function's short name to its weight name (in the ini file). The `mert-moses.pl` script will query the decoder to find the other information that it needs.

Subsection last modified on September 12, 2012, at 12:56 PM

6.7 Adding Sparse Feature Functions

Moses allows for sparse feature functions, i.e., feature functions that have a large, maybe unbounded, set of features, of which only a small subset applies to a given hypothesis.

To give an example: In addition to a regular n-gram language model, we could introduce a discriminative bigram language model that discounts or promotes hypotheses that contain specific bigrams. Each bigram in this feature function is its own feature with its own feature weight.

These features cannot be tuned with MERT, but Moses has several other suitable tuning (page **??**) methods.

The incorporation of sparse features into the training pipeline[21] is ongoing.

6.7.1 Implementation

For basics, please refer to the respective section on Feature Functions (page **??**).

Header

```
class PhraseLengthFeature : public StatelessFeatureFunction {
public:
PhraseLengthFeature():
StatelessFeatureFunction("phrl")
{}
```

instead of

```
class GlobalLexicalModel : public StatelessFeatureFunction {
[...]
GlobalLexicalModel(const std::string &filePath,
const std::vector< FactorType >& inFactors,
const std::vector< FactorType >& outFactors);
```

hmm.... Not sure why passing the description is necessary. *because it saves you having to implement GetScoreProducerDescription() - ed*

Setting feature values

As for the basic feature functions, a `Evaluate` function needs to be defined. In the sparse feature branch the stateless version has been updated to facilitate pre-computation and to prevent the feature from interfering with recombination.

Stateless feature functions are called with

```
Evaluate(const TranslationOption& translationOption,
const InputType& inputType,
const WordsBitmap& coverageVector,
ScoreComponentCollection* accumulator)
```

[21] http://www.statmt.org/moses/?n=Moses.SparseFeatureTraining

Stateful feature functions are called with

```
Evaluate(const Hypothesis& cur_hypo,
const FFState* prev_state,
ScoreComponentCollection* accumulator) const
```

and have to return a feature function state (FFState*).

The evaluation function computes what it needs to compute and then updates weights with a call such as:

```
accumulator->PlusEquals(this,name,1);
```

where name is a string with the feature name and 1 is the addition to the feature value.

Pre-computation

If a stateless feature does not depend on the coverage vector (i.e. most features) then its values should be *pre-computed* and stored in the TranslationOptionsCollection. This is achieved by overriding the ComputeValueInTranslationOption() to return true.

Additionally, if a stateless feature function does not depend on any properties of the source sentence then it may be pre-computed and stored in the phrase table. There are hooks to add this to EMS (page **??**), but they are currently undergoing redevelopment and are not documented. Sparse features can be added to the phrase table by adding an extra field at the end of the line, with the feature vector in the format name1 value1 name2 value2 .... Binarisation of the table will preserve the feature vector.

6.7.2 Weights

There is no need to define a switch weight-x for the feature function. Each feature of the feature function has its own named weight, which is a concatenation of the short name of the feature function, and underscore (_) and its individual name for which the feature function sets.

These weights are placed into a weight file which is specified with the switch -weight-file. For example, the target bigram feature weights (feature function short name dlmb for discriminative language model, bigrams) may have weights defined in lines such this:

```
dlmb_of:the 0.1
dlmb_in:the -0.1
dlmb_the:way 0.2
```

Features that do not have weights that are defined in this file are set to 0.

Subsection last modified on October 25, 2012, at 05:07 PM

6.8 Regression Testing

6.8.1 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.

6.8.2 Test suite

The following regression tests are currently implemented (and more have been added since this list was written):

- `basic-surface-only` Tests basic translation, compares output strings and probabality scores.
- `basic-surface-binptable` Tests binary phrase table
- `consensus-decoding-surface` Basic test of consensus decoding
- `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

6.8.3 Running the test suite

Download the regression tests

```
git clone https://github.com/moses-smt/moses-regression-tests.git
```

From the Moses root, run

```
./bjam --with-irstlm=/path/to/irst --with-srilm=/path/to/sri --with-regtest=/path/to/moses-regression-tests -j8
```

This will run the regression tests in parallel (-j8) so be sure to set a number of CPUs that your machine can handle.

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.

6.8.4 Running an individual test

You can run a specific test by providing the name followed by ".passed"

```
./bjam --with-irstlm=/path/to/irst --with-srilm=/path/to/sri --with-regtest=/path/to/moses-regression-tests mert.basic.passed
```

The test name is the same as the directory name in /path/to/moses-regression-tests/tests .

6.8.5 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 values are then compared to a ground truth, which was established either by hand, from a prior moses run, or from a pharaoh run.

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).

6.8.6 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.

Subsection last modified on October 20, 2012, at 09:36 PM

7

Reference

7.1 Frequently Asked Questions

7.1.1 Content

- My system is taking a really long time to translate a sentence. What can I do to speed it up ? (page **??**)
- The system runs out of memory during decoding. (page **??**)
- I would like to point out a bug / contribute code. (page **??**)
- How can I get an updated version of Moses ? (page **??**)
- When is version 1/beta of Moses coming out? (page **??**)
- I'm an undergrad/masters student looking for a project in SMT. What should I do? (page **??**)
- What do the 5 numbers in the phrase table mean? (page **??**)
- What OS does Moses run on? (page **??**)
- Can I use Moses on Windows ? (page **??**)
- Do I need a computer cluster to run experiments? (page **??**)
- I've compiled Moses, but it segfaults when running. (page **??**)
- How do I add a new feature function to the decoder? (page **??**)
- Compiling with SRILM or IRSTLM produces errors. (page **??**)
- I'm trying to use Moses to create a web page to do translation. (page **??**)
- How can a create a system that translate both ways, ie. X-to-Y as well as Y-to-X ? (page **??**)
- PhraseScore dies with signal 11 - why? (page **??**)
- Does Moses do Hierarchical decoding, like Hiero etc? (page **??**)
- Can I use Moses in proprietary software ? (page **??**)
- GIZA++ crashes with error "parameter 'coocurrencefile' does not exist." (page **??**)

• Running regenerate-makefiles.sh gives me lots of errors about *GREP and *SED macros (page **??**)

• Running training I got the following error "*** buffer overflow detected ***: ../giza-pp/GIZA++-v2/GIZA++ terminated" (page **??**)

• I retrained my model and got different bleu scores. Why? (page **??**)

• I specified ranges for mert weights, but it returned weights which are outwith those ranges (page **??**)

• Who do I ask if my question hasn't been answered by this FAQ? (page **??**)

7.1.2 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.

7.1.3 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 **??**) tell you how to binarize the models.

7.1.4 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 github account here[1]

Then ask one of the project admins to add you to the Moses project. The admins are currently

- Barry Haddow
- Hieu Hoang
- Nicola Bertoldi
- Ondrej Bojar
- Kenneth Heafield

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.

[1] https://github.com/

7.1.5 How can I get an updated version of Moses ?

The best way is using git.

From the command line, type

```
git pull
```

Or use whatever GUI client you have.

7.1.6 When is version 1/beta of Moses coming out?

See Releases (page **??**)

7.1.7 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.

7.1.8 What do the 5 numbers in the phrase table mean?

See the section on phrase scoring (page **??**)

7.1.9 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 are regularly run on Linux (32 and 64-bits), and occasionally on Mac (Intel). The whole of the Moses pipeline should also run on Windows under Cygwin, however, this has not been confirmed. If you are able to run under Windows/Cygwin, please let us know and we can update this FAQ.

When running on non-Linux platforms, beware of the following issues:

- File system case-sensitivity
- `zcat`, `gzip` command line programs missing
- Old GIZA++ versions 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 and Intel Mac.

7.1.10 Can I use Moses on Windows ?

Yes. Moses compiles and runs in cygwin exactly the same way as on linux

There are a proviso though:

Cygwin is 32-bit, even on 64 bit windows. The binary language models (kenlm, irstlm) need 64 bit to work with lm larger than about 2gb. This is the same as for 32 bit linux.

7.1.11 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.

7.1.12 I've compiled Moses, but it segfaults when running.

Moses shouldn't segfault, so the Moses developers would like to hear about it.

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 and XCode 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.

7.1.13 How do I add a new feature function to the decoder?

This is now documented in its own section (page **??**).

7.1.14 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.

SRI or IRST LM both have their own mailing list where you can ask questions if you have problem compiling them. See here for details:

- SRILM[2]

[2] http://www.speech.sri.com/mailman/listinfo/srilm-user

- IRSTLM[3]

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.

7.1.15 I'm trying to use Moses to create a web page to do translation.

There's a subproject in moses, in `contrib/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 !

7.1.16 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.

7.1.17 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
```

[3] https://list.fbk.eu/sympa/subscribe/user-irstlm

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-model.perl ... --first-step 5
```

7.1.18 Does Moses do Hierarchical decoding, like Hiero etc?

Yes. Check the Syntax Tutorial (page **??**).

7.1.19 Can I use Moses in proprietary software ?

Moses is licensed under the LGPL. See here[4] for a thorough explanation of what this means.

Basically, if you're just using moses unchanged, there's no license issues. You can also use the moses library (`libmoses.a`) in your own applications. But if you want to distribute a modified version of moses, you have to distribute the source code to the modifications.

7.1.20 GIZA++ crashes with error "parameter 'coocurrencefile' does not exist."

You have a version of GIZA++ which doesn't support cooccurrence files. To add support for cooccurrence files, you need to edit the GIZA++ Makefile and add the flag `-DBINARY_SEARCH_FOR_TTABLE` to `CFLAGS_OPT`. Then you should rebuild GIZA++.

7.1.21 Running regenerate-makefiles.sh gives me lots of errors about *GREP and *SED macros

You shouldn't be running this script. Moses moved from autotools to bjam in Autumn 2011.

[4] http://www.gnu.org/licenses/gpl-faq.html

7.1.22 Running training I got the following error "*** buffer overflow detected ***: ../giza-pp/GIZA++-v2/GIZA++ terminated"

This error occurs during the word alignment step and is related to GIZA++, and not directly to the Moses Toolkit. Neverthless, the solution is described here[5].

7.1.23 I retrained my model and got different bleu scores. Why?

In general, Machine Translation training is non-convex. this means that there are multiple solutions and each time you run a full training job, you will get different results. In particular, you will see different results when running Giza++ (any flavour) and MERT.

The best way to deal with this (and most expensive) would be to run the full pipe-line, from scratch and multiple times. This will give you a feel for variance –differences in results. In general, variance arising from Giza++ is less damaging than variance from MERT.

To reduce variance it is best to use as much data as possible at each stage. It is possible to reduce this variability by using better machine learning, but in general it will always be there.

Another strategy is to fix everything once you have a set of good weights and never rerun MERT. Should you need to change say the language model, you will then manually alter the associated weight. This will mean stability, but at the obvious cost of generality. it is also ugly.

See Clark et al.[6] for a discussion of some of these issues.

7.1.24 I specified ranges for mert weights, but it returned weights which are outwith those ranges

The ranges that you pass to `mert-moses.pl` (using the `-range` argument) are only used in the random restarts, so serve to guide mert rather than restrict it.

7.1.25 Who do I ask if my question hasn't been answered by this FAQ?

Search the mailing list archive[7]. If you still don't find the answer, then send questions to the mailing list 'moses-support'. However, you have to sign up[8] before emailing.

Subsection last modified on November 04, 2012, at 07:07 AM

[5] http://www.statmt.org/moses/?n=Moses.GizappBufferOverflow

[6] http://www.cs.cmu.edu/~jhclark/pubs/significance.pdf

[7] http://blog.gmane.org/gmane.comp.nlp.moses.user

[8] http://mailman.mit.edu/mailman/listinfo/moses-support

7.2 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
- `-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'

Subsection last modified on January 29, 2010, at 11:43 AM

7.3 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), `grow-diag-final-and`, `srctotgt`, `tgttosrc`
- `-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` – specifies which reordering models to train using a comma-separated list of config-strings, see FactoredTraining.BuildReorderingModel (page **??**). (default distance)
- `-reordering-smooth` – specifies the smoothing constant to be used for training lexicalized reordering models. If the letter "u" follows the constant, smoothing is based on actual counts. (default 0.5)
- `-alignment-factors` –
- `-translation-factors` –
- `-reordering-factors` –
- `-generation-factors` –
- `-decoding-steps` –

7.3.1 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-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

7.3.2 Factored Translation Model Settings

More on factored translation models in the Overviev (page **??**).

Summary

- `-alignment-factors` –
- `-translation-factors` –
- `-reordering-factors` –
- `-generation-factors` –
- `-decoding-steps` –

7.3.3 Lexicalized Reordering Model

More on lexicalized reording on the description of Training step 7: build reordering model (page **??**).

Summary

- `-reordering` –
- `-reordering-smooth` –

7.3.4 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-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)

7.3.5 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-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)`

7.3.6 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)

7.3.7 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)

7.3.8 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-model.perl [...] --giza-option m1=4,m2=0,mh=4,m3=0,m4=3
```

Summary

- -giza-option – additional options for GIZA++ training

7.3.9 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)

Subsection last modified on September 05, 2011, at 12:40 PM