NiuTrans Open Source

Statistical Machine Translation System

Brief Introduction and User Manual

Natural Language Processing Lab
Northeastern University, China
niutrans@mail.neu.edu.cn
http://www.nlplab.com

Version 1.1.0 Beta

The document was prepared by the following:

Tong Xiao
xiaotong@mail.neu.edu.cn
co-PI

Jingbo Zhu
zhujingbo@mail.neu.edu.cn
co-PI

Hao Zhang
zhanghao1216@gmail.com
core developer

Qiang Li
liqiangneu@gmail.com
core developer

For any questions, please feel free to mail to us (niutrans@mail.neu.edu.cn)

Table of Contents

Chapter 1

Introduction

1.1 Welcome to NiuTrans

NiuTrans is an open-source statistical machine translation system developed by the Natural Language Processing Group at Northeastern University, China. The NiuTrans system is fully developed in C++ language. So it runs fast and uses less memory. Currently it supports (hierarchical) phrase-based and syntax-based models, and provides easy-to-use APIs for research-oriented experiments.

This document serves as a user manual for all the functions of the NiuTrans system. First, it introduces the basic features of NiuTrans and some necessary instructions for installing it (Section 1). Then, a brief manual is presented in Section 2 to provide a very brief usage of the system. Section 3 and Section 4 give more details about the phrase-based and syntax-based engines involved in NiuTrans, including background of phrase-based and syntax-based MT and a step-by-step manual. Beyond this, a number of interesting features are presented in Section 5 for advanced users. In addition, some frequently-asked questions and their answers are presented in Section 6.

If you are a "lazy" guy and do not want to go deep into the details of the underlying methods, reading Section 1 and Section 2 is enough. However, if you are interested in the features provided within NiuTrans and would like to learn more about how to set-up a better translation system for your task, it is suggested to go through the whole document, especially Sections 3-5. We think it would help.

For any questions about NiuTrans, please e-mail to us (niutrans@mail.neu.edu.cn) directly.

1.2 How to Cite NiuTrans

If you use NiuTrans in your research and would like to acknowledge this project, please cite the following paper which will appear in the system demonstration session of the 50th Annual Meeting of the Association

for Computational Linguistics (ACL).

> Tong Xiao, Jingbo Zhu, Hao Zhang and Qiang Li. 2012. NiuTrans: An Open Source Toolkit for Phrase-based and Syntax-based Machine Translation. In *Proc.* of ACL System Demonstrations, pages 19-24.

1.3 Related Work

To date, several open-source SMT systems have been developed, showing state-of-the-art performance for many tasks, such as Chinese-English and Arabic-English translation. Some of these systems focus on phrased-based models (such as Moses) which have been widely-used in the community for years, while others try to use hierarchical models (such as Joshua) to handle the syntactic structure movement between languages. Although these systems and approaches are of competitive translation quality, they have different focuses as well as different strengths and weaknesses. The following is a brief review of current open-source SMT systems.

- **Moses**[1]. Moses is a pioneer SMT system developed (mainly) by the SMT group at the University of Edinburgh [Koehn et al., 2007]. The newest version of Moses supports a number of features. For example, it supports both the phrase-based and syntax-based models (from phrase/rule extraction to decoding). Also, it offers the factored translation model which enables the use of various information at different levels. Moreover, confusion networks and word lattices are allowed to be used as input to alleviate errors in the 1-best output of ambiguous upstream systems. In addition, the Moses package provides many useful scripts and tools to support additional features.

- **Joshua**[2]. Joshua is another state-of-the-art open-source SMT system developed at the Center for Language and Speech Processing at the Johns Hopkins University [Li et al., 2009]. The underlying model used in Joshua is the hierarchical phrase-based model proposed in [Chiang, 2005]. In addition to the base model, it provides several interesting features, such as decoding with (syntax-annotated) SCFGs, variational decoding and parallel training with map-reduce. As Joshua is implemented in Java language, it has good extensibility and portability for running/development on different platforms. Also, the use of Java provides an easy way (compared to C/C++) to experiment with new ideas and advance current state-of-the-art results.

- **SilkRoad**[3]. SilkRoad is a phrase-based SMT system developed by five universities and research institutions in China (CAS-ICT, CAS-IA, CAS-IS, XMU and HIT). The SilkRoad system is the first open-source SMT system in the asian area, with a primary goal of supporting Chinese-Foreign translation as well as the translation for other language pairs. It has several useful components, such as word segmentation module, which make users can easily build Chinese-Foreign translation

[1] http://www.statmt.org/moses/
[2] http://joshua.sourceforge.net/Joshua/Welcome.html
[3] http://www.nlp.org.cn/project/project.php?proj_id=14

systems. Moreover, multiple decoders and rule extractors are supported in SilkRoad and provides diverse options for experimentation with different combinations of subsystems.

- **SAMT**[4]. SAMT is a syntax-augmented SMT system developed by the MT group at Carnegie Mellon University [Zollmann and Venugopal, 2006]. SAMT induces translation rules using the target-trees, while does not strictly respect target-syntax during decoding. The highlight of SAMT is that it offers an simple but effective way to make use of syntactic information in SMT and shows good results in several tasks, even outperforms the hierarchical phrase-based counterparts in some cases. As SAMT is implemented in hadoop, it can benefit from the distributed processing of large data sets across clusters of computers.

- **cdec**[5]. cdec is a powerful decoder developed by Chris Dyer and his collaborators [Dyer et al., 2010]. The major feature of cdec is that it uses a unified internal representation of translation models and provides a framework for experimenting with various models and algorithms for structure prediction problems. So cdec can also be used as an aligner or a more general learning framework for SMT. Moreover, the cdec decoder is very fast due to the effective use of C++ in development.

- **Phrasal**[6]. Phrasal was developed by the Stanford Natural Language Processing Group [Cer et al., 2010]. In addition to the traditional phrase-based model, it also supports the non-hierarchical phrase-based model which extends phrase-based translation to phrasal discontinues translation. In this way, it could provide better generalization on unseen data, even handle the cases missed in hierarchical models (as in Joshua).

- **Jane**[7]. Jane is another C++ implementation of the phrase-based and hierarchical phrase-based models. It was developed by the Human Language Technology and Pattern Recognition Group at RWTH Aachen University [Vilar et al., 2010]. Jane supports many interesting features (such as MIRA for weight tuning) and shows competitive result in several tasks.

The above systems offer good choices for building SMT systems with low prices. Also, they are very nice references for the development of the NiuTrans system.

1.4 Why NiuTrans

While SMT has been studied for decades and several open-source systems have been developed, we believe that NiuTrans is still promising because it has the following features:

- NiuTrans is written in C++ and optimized in several ways. So it is fast.

- NiuTrans is easy-to-use. All you need is running a few scripts. Also, it offers a number of APIs for feature engineering.

[4]http://www.cs.cmu.edu/ zollmann/samt/
[5]http://cdec-decoder.org/index.php?title=Main_Page
[6]http://nlp.stanford.edu/phrasal/
[7]http://www-i6.informatik.rwth-aachen.de/jane/

- A compact but efficient n-gram Language Model (LM) is embedded in NiuTrans. It does not need external support from other softwares (such as SRILM and IRST's language modeling toolkits)
- NiuTrans provides a unified framework that support most of current state-of-the-art models, including
  - Phrase-based model
  - Hierarchical phrase-based model
  - Syntax-based model (string-to-tree/tree-to-string/tree-to-tree)

  By using NiuTrans, ones can conduct empirical testing/comparsison with various approaches and algorithms under the same implementation/experimental environment.
- NiuTrans allows the decoding of input string (as in parsing) or input tree (as in tree-parsing).
- NiuTrans has competitive performance for Chinese-Foreign translation tasks.

1.5 Open Source License

The NiuTrans system is open-source available under the GNU General Public License[8].

1.6 Required Softwares

To use NiuTrans, some softwares should be prepared and installed on your computer in advance.

For Windows users, Visual Studio 2008, Cygwin[9], and perl[10] (version 5.10.0 or higher) are required. It is suggested to install cygwin under path "C:\" by default.

For Linux users, gcc (version 4.1.2 or higher), g++ (version 4.1.2 or higher), GNU Make (version 3.81 or higher) and perl (version 5.8.8 or higher) are required.

NOTE: 2GB memory and 10GB disc space is a minimal requirement for running the system. Of course, more memory and disc space are helpful if the system is trained on large corpora. To support large models (such as the n-gram LM trained on GIGAWORD corpus), 64bit OS is recommended.

1.7 Installation

The installation of the NiuTrans system is simple. Please unpack the downloaded package (suppose that the target directory is "NiuTrans") and use the following instructions to build the system[11].

[8] http://www.gnu.org/licenses/gpl-2.0.html
[9] http://www.cygwin.com/
[10] http://www.activestate.com/activeperl
[11] it is suggested to run NiuTrans on Linux systems

1.7.1 For Windows Users

Command

```
- open "NiuTrans.sln" in "NiuTrans\src\"
- set configuration mode to "Release"
- set platform mode to "Win32" (for 32bit OS) or "x64" (for 64bit OS)
- build the whole solution
```

Then, you will find that all binaries are generated in "NiuTrans\bin\".

1.7.2 For Linux Users

Command

```
$ cd NiuTrans/src/
$ chmod a+x install.sh
$ ./install.sh -m32 (for 32bit OS) or ./install.sh (for 64bit OS)
$ source ~/.bashrc
```

Then, you will find that all binaries are generated in "NiuTrans/bin/".

Chapter 2

Quick Walkthrough

2.1 Data Preparation

The NiuTrans system is a "data-driven" system. So it requires "data" for training and/or tuning the system. Some necessary data files should be prepared before running the system.

1. **Training data**: bilingual sentence-pairs and word alignments.
2. **Tuning data**: source sentences with one or more reference translations.
3. **Test data**: some new sentences.
4. **Evaluation data**: reference translations of test sentences.

The NiuTrans package offers some sample files for experimenting with the system and studying the format requirement. They are placed in "NiuTrans/sample-data/sample-submission-version".

Sample Data (NiuTrans/sample-data/sample-submission-version)

```
- TM-training-set/chinese.txt            ▷ source sentences
- TM-training-set/english.txt            ▷ target sentences (case-removed)
- TM-training-set/Alignment.txt          ▷ word alignments
- LM-training-set/e.lm.txt               ▷ monolingual corpus for training LM
- Dev-set/Niu.dev.txt                    ▷ tuning data
- Test-set/Niu.test.txt                  ▷ test data
- Reference-for-evaluation/Niu.test.reference ▷ references of the test sentences
- description-of-the-sample-data         ▷ a description of the sample data
```

Please unpack "NiuTrans/sample-data/sample.tar.gz", and refer to "description-of-the-sample-data" for more information about data format

In the following, the above data files will be used to illustrate how to run the NiuTrans system (e.g. training MT models, tuning feature weights, decoding new sentences and etc.). For convenience, the phrase-

based engine (NiuTrans.Phrase) is used in the illustration. For other translation engines, see Section 4 for more details.

2.2 Training

In NiuTrans, the standard training procedure involves two steps.

- learning translation model
- learning n-gram language model

Step 1: To obtain the translation model, it is required to extract phrase translations and estimate the associated scores (features) from bilingual sentence-pair. This step can be trivially done by using script "scripts/NiuTrans-phrase-train-model.pl"

Command

```
$ cd NiuTrans/sample-data/
$ tar xzf sample.tar.gz
$ cd ../
$ mkdir work/model.phrase/ -p
$ cd scripts/
$ perl NiuTrans-phrase-train-model.pl \
      -tmdir   ../work/model.phrase/ \
      -s       ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -t       ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -a       ../sample-data/sample-submission-version/TM-training-set/Alignment.txt
```

where

`-tmdir` specifies the target directory for generating various tables and model files.

`-s`, `-t` and `-a` specify the source-language side of the training data (one sentence per line).

Note: Please enter the "scripts" directory before running the script "NiuTrans-phrase-train-model.pl".

Output: The output of this step is three files placed in "NiuTrans/work/model.phrase/":

Output (NiuTrans/work/model.phrase/)

```
- me.reordering.table           ▷ ME reorder model
- msd.reordering.table          ▷ MSD reorder model
- phrase.translation.table      ▷ phrase translation model
```

Step 2: Then, the n-gram language model is trained on the monolingual corpus, as follows:

Command

```
$ cd ../
$ mkdir work/lm/
$ cd scripts/
$ perl NiuTrans-training-ngram-LM.pl \
      -corpus    ../sample-data/sample-submission-version/LM-training-set/e.lm.txt \
      -ngram     3 \
      -vocab     ../work/lm/lm.vocab \
      -lmbin     ../work/lm/lm.trie.data
```

where

`-corpus` specifies the training data file (i.e., a collection of target-language sentences. one sentence per line).

`-ngram` specifies the order of n-gram LM. E.g. "-ngram 3" indicates a 3-gram language model.

`-vocab` specifies where the target-side vocabulary is generated.

`-lmbin` specifies where the language model file is generated.

Output: The output of this step is two files placed in "NiuTrans/work/lm/".

Output (NiuTrans/work/model/)

```
- lm.vocab                    ▷ target-side vocabulary
- lm.trie.data                ▷ binary-encoded language model
```

2.3 Generating Configuration File for Decoding

Once the model is learned, it is required to create a configuration file for the following decoding process. To do this, you can run the following script:

Command

```
$ cd scripts/
$ perl NiuTrans-phrase-generate-mert-config.pl \
      -tmdir   ../work/model.phrase/ \
      -lmdir   ../work/lm/ \
      -ngram   3 \
      -o       ../work/NiuTrans.phrase.user.config
```

where

`-tmdir` specifies the directory that holds the translation table and the reordering model files.

`-lmdir` specifies the directory that holds the n-gram language model and the target-side vocabulary.

`-ngram` specifies the order of n-gram language model.

`-o` specifies the output (i.e., a config file).

Output: The output is a config file that is generated and placed in "NiuTrans/work/":

Output (NiuTrans/work/)

```
- NiuTrans.phrase.user.config        ▷ configuration file for decoding
```

2.4 Weight Tuning

Next, the feature weights are optimized on the development data-set. In NiuTrans, Minimum Error Rate Training (MERT) is used as the default optimization algorithm for weight tuning. The MER training can be executed using the following instructions:

Command

```
$ perl NiuTrans-phrase-mert-model.pl \
      -dev    ../sample-data/sample-submission-version/Dev-set/Niu.dev.txt \
      -c      ../work/NiuTrans.phrase.user.config \
      -nref   1 \
      -r      3 \
      -l      ../work/mert-model.log
```

where

`-dev` specifies the development set for weight tuning.

`-c` specifies the configuration file generated in the previous steps.

`-nref` specifies how many reference translations per source-sentence are provided.

`-r` specifies how many rounds MERT performs (by default, 1 round = 15 MERT iterations).

`-l` specifies the log file generated by the MERT program.

Output: The optimized feature weights are recorded in the configuration file "NiuTrans/work/NiuTrans.phrase.user.config". They will then be used in decoding test sentences.

2.5 Testing

If all the above steps are finished, the training of translation system is over. The learned model (including translation table, n-gram LM and etc.) is then used to translate new sentences. Decoding new sentences is trivial in NiuTrans. Please do it as follows:

Command

```
$ perl NiuTrans-phrase-decoder-model.pl \
      -test    ../sample-data/sample-submission-version/Test-set/Niu.test.txt \
      -c       ../work/NiuTrans.phrase.user.config \
      -output  1best.out
```

where

`-test` specifies the test data-set (one sentence per line).

`-c` specifies the configuration file.

`-output` specifies the file of translation result (the result is dumped to "stdout" if this option is not specified).

Output: 1-best translation of the test sentences. See file "1best.out" in "NiuTrans/scripts/".

Output (NiuTrans/scripts/)

```
- 1best.out        ▷ 1-best translation of the test sentences
```

2.6 Evaluation

Last, the result is evaluated in terms of BLEU score. The following scripts can help you do this (mteval-v13a.pl is required for BLEU calculation[1]).

Command

```
$ perl NiuTrans-generate-xml-for-mteval.pl \
      -1f    1best.out \
      -tf    ../sample-data/sample-submission-version/Reference-for-evaluation/Niu.test.reference \
      -rnum  1
$ perl mteval-v13a.pl \
      -r     ref.xml \
      -s     src.xml \
      -t     tst.xml
```

where

`-1f` specifies the file of the 1-best translations of the test data-set.

`-tf` specifies the file of the source sentences and their reference translations of the test data-set.

`-r` specifies the file of the reference translations.

[1] ftp://jaguar.ncsl.nist.gov/mt/resources/mteval-v13a.pl

`-s` specifies the file of source sentence.

`-t` specifies the file of (1-best) translations generated by the NiuTrans system.

Output: The IBM-version BLEU score is displayed on the screen.

Note: running script mteval-v13a.pl replies on the package XML::Parser[2]. If XML::Parser is not installed on your system, please use the following commands to install it.

Command

```
$ su root
$ tar xzf XML-Parser-2.41.tar.gz
$ cd XML-Parser-2.41/
$ perl Makefile.PL
$ make install
```

[2]http://search.cpan.org/ toddr/XML-Parser-2.41/Parser.pm

Chapter 3

NiuTrans.Phrase - A Phrase-Based Translation Engine

Like other SMT packages, the phrase-based model is supported in NiuTrans. The basic idea of phrase-based MT is to decompose the translation process into a sequence of phrase compositions, and gives an estimation of translation probability using various features associated with the underlying derivation of phrases. Due to its simplicity and strong experimental results, phrase-based SMT has been recognized as one of the most successful SMT paradigms and widely used in various translation tasks.

The development of the phrase-based engine in NiuTrans (called NiuTrans.Phrase) started from the early version of a competition system in CWMT2009 [Xiao et al., 2009]. Over the past few years, this system has been advanced in several MT evaluation tasks such as CWMT2011 [Xiao et al., 2011b] and NICIR-9 PatentMT [Xiao et al., 2011a]. Currently NiuTrans.Phrase supports all necessary steps in the standard phrase-based MT pipeline, with a extension of many interesting features. In the following parts of this section, NiuTrans.Phrase will be described in details, including a brief introduction of the background knowledge (Section 3.1) and a step-by-step manual to set-up the system (Sections 3.2∼3.6).

Note: Section 3.1 is for the readers who are not familiar with (statistical) machine translation. If you have basic knowledge of SMT, please skip Section 3.1 and jump to Section 3.2 directly.

3.1 Background

3.1.1 Mathematical Model

The goal of machine translation is to automatically translate from one language (a source string $\mathbf{s}$) to another language (a target string $\mathbf{t}$). In SMT, this problem can be stated as: we find a target string $\mathbf{t}^*$ from all possible translations by the following equation:

$$\mathbf{t}^* = \arg\max_{\mathbf{t}} \Pr(\mathbf{t}|\mathbf{s}) \tag{3.1}$$

where $\Pr(\mathbf{t}|\mathbf{s})$ is the probability that $\mathbf{t}$ is the translation of the given source string $\mathbf{s}$. To model the

posterior probability $\Pr(\mathbf{t}|\mathbf{s})$, the NiuTrans system utilizes the log-linear model proposed by Och and Ney [2002]:

$$\Pr(\mathbf{t}|\mathbf{s}) = \frac{\sum_{i=1}^{M} \lambda_i \cdot h_i(\mathbf{s},\mathbf{t})}{\sum_{\mathbf{t}'} \sum_{i=1}^{M} \lambda_i \cdot h_i(\mathbf{s},\mathbf{t}')} \tag{3.2}$$

where $\{h_i(\mathbf{s},\mathbf{t})|i = 1, ..., M\}$ is a set of features, and λ_i is the feature weight corresponding to the i-th feature. $h_i(\mathbf{s},\mathbf{t})$ can be regarded as a function that maps each pair of source string $\mathbf{s}$ and target string $\mathbf{t}$ into a non-negative value, and λ_i can be regarded as the contribution of $h_i(\mathbf{s},\mathbf{t})$ to $\Pr(\mathbf{t}|\mathbf{s})$. Ideally, λ_i indicates the pairwise correspondence between the feature $h_i(\mathbf{s},\mathbf{t})$ and the overall score $\Pr(\mathbf{t}|\mathbf{s})$. A positive value of λ_i indicates a correlation between $h_i(\mathbf{s},\mathbf{t})$ and $\Pr(\mathbf{t}|\mathbf{s})$, while a negative value indicates an inversion correlation.

In this document, u denotes a model that has M fixed features $\{h_1(\mathbf{s},\mathbf{t}), ..., h_M(\mathbf{s},\mathbf{t})\}$, $\lambda = \{\lambda_1, ..., \lambda_M\}$ denotes the M parameters of u, and $u(\lambda)$ denotes the SMT system based on u with parameters λ. In a general pipeline of SMT, λ is learned on a tuning data-set to obtain an optimized weight vector λ^* as well as an optimized system $u(\lambda^*)$. To learn the optimized weight vector λ^*, λ is usually optimized according to a certain objective function that 1) takes the translation quality into account; 2) and can be automatically learned from MT outputs and reference translations (or human translations). For example, we can use BLEU [Papineni et al., 2002], a popular metric for evaluating translation quality, to define the error function and learn optimized feature weights using the minimum error rate training method.

In principle, the log-linear model can be regarded as an instance of the discriminative model which has been widely used in NLP tasks [Bergert et al., 1996]. In contrast with modeling the problem in a generative manner [Brown et al., 1993], discriminative modeling frees us from deriving the translation probability for computational reasons and provides capabilities to handle the features that are able to distinguish between good and bad translations [Lopez, 2008]. In fact, arbitrary features (or sub-models) can be introduced into the log-linear model, even if they are not explained to be well-formed probabilities at all. For example, we can take both phrase translation probability and phrase count (i.e., number of phrases used in a translation derivation) as features in such a model. As the log-linear model has emerged as a dominant mathematical model in SMT in recent years, it is chosen as the basis of the NiuTrans system.

3.1.2 Translational Equivalence Model

Given a source string $\mathbf{s}$ and a target string $\mathbf{t}$, MT systems need to model the translational equivalence between them. Generally speaking, a translational equivalence model is a set of possible translation steps (units) that are involved in transforming $\mathbf{s}$ to $\mathbf{t}$. Many ways can be considered in defining the translational equivalence model. For example, in word-based models [Brown et al., 1993], translation units are defined on individual word-pairs, and the translation process can be decomposed into a sequence of compositions of word-pairs.

The phrase-based SMT extends the idea of word-based translation. It discards the restriction that a translation unit should be on word-level, and directly defines the unit of translation on any sequence of words (or *phrases*). Therefore it could easily handle the translations inherent in phrases (such as local reordering), and does not rely on the modeling of *null-translation* and $fertility$ that are somewhat thorny

in word-based models. Under such a definition, the term "phrase" does not have a linguistic sense, but instead focuses more on a "n-gram" translation model. The phrase-based model also allows free boundaries of phrases and thus defers the explicit tokenization step which is required in some languages, such as Chinese and Japanese.

More formally, we denote the input string $\mathbf{s}$ as a sequence of source words $s_1...s_J$, and the output string $\mathbf{t}$ as a sequence of target words $t_1...t_I$. Then we use $\bar{s}[j_1, j_2]$ (or $\bar{s}$ for short) to denote a source-language phrase spanning from position j_1 to position j_2. Similarly, we can define $\bar{t}[i_1, i_2]$ (or $\bar{t}$ for short) on the target-language side. In the phrase-based model, the following steps are required to transform $\mathbf{s}$ into $\mathbf{t}$.

1. Split $\mathbf{s}$ into a sequence of phrases $\{\bar{s}_1...\bar{s}_K\}$, where K is the number of phrases.
2. Replace each $\bar{s}_j \in \{\bar{s}_1...\bar{s}_K\}$ with a target phrase $\bar{t}_i$. Generally a one-to-one mapping is assumed in phrase-based models. So this step would result in exact K target phrase(s) $\{\bar{t}_1...\bar{t}_K\}$.
3. Permute the target phrases $\{\bar{t}_1...\bar{t}_K\}$ in an appropriate order.

The above procedure implies two fundamental problems in phrase-based SMT.

- How should phrase translations be learned?
- How should target phrases be permuted?

Although phrase translations can be in principle learned from anywhere, current phrase-based systems require a process of extracting them from bilingual corpus. In this way, the first problem mentioned above is also called *phrase extraction*. The second problem is actually identical to the one we have to deal with in word-based models, and thus called the *reordering* problem.

Both the two problems are addressed in NiuTrans. For phrase extraction, a standard method [Koehn et al., 2003] is used to extract phrase translations from word-aligned bilingual sentence-pairs. For the reordering problem, the ITG [Wu, 1997] constraint is employed to reduce the number of possible reordering patterns, and two reordering models are adopted for detailed modeling. In the following two sections, these methods will be described in more detail.

3.1.3 Phrase Extraction

In Koehn et al. [2003]'s model, it is assumed that words are initially aligned (in some way) within the given sentence-pair. As a consequence explicit internal alignments are assumed within any phrase-pairs. This means that, before phrase extraction, ones need a word alignment system to obtain the internal connections between the source and target sentences. Fortunately, several easy-to-use word alignment toolkits, such as GIZA++[1], can do this job. Note that, in NiuTrans, word alignments are assumed to be prepared in advance. We do no give further discussion on this issue in this document.

The definition of phrase-pairs are pretty simple: given a source string, a target string and the word alignment between them, valid phrase pairs are defined to those string pairs which are consistent with the

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

word alignment. In other words, if there is a alignment link outside a given phrase-pair, the extraction of the phrase-pair would be blocked. Figure 3.1 illustrates this idea with some sample phrases extracted from a sentence-pair.

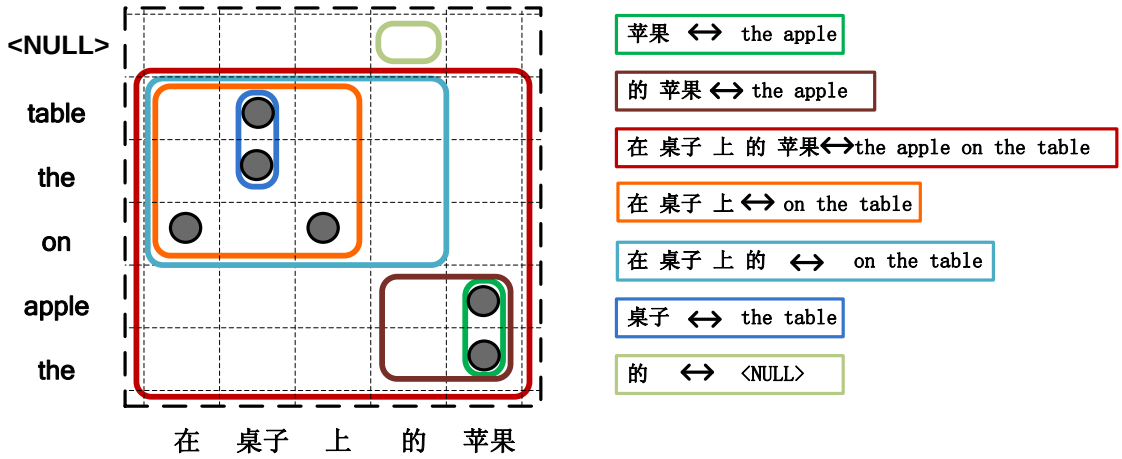


Figure 3.1. Sample phrase-pairs extracted from a word-aligned sentence-pair. Note that explicit word deletion is allowed in NiuTrans.

To extract all phrase-pairs from given source sentence and target sentence, a very simple algorithm could be adopted. Its basic idea to enumerate all source-phrases and target-phrases and rule out the phrase-pairs that violate the word alignment. The following pseudocode (Figure 3.2) summarizes the rule extraction algorithm used in NiuTrans. It is worth noting that this algorithm has a complexity of $O(J \cdot I \cdot ls_{max}^2 \cdot lt_{max}^2)$ where ls_{max} and lt_{max} are the maximum lengths of source and target phrases, respectively. Setting ls_{max} and lt_{max} to very large numbers is not helpful on test sentences, even is not practical to real-word systems. In most cases, only those (relatively) small phrases are considered during phrase extraction. For example, it has been verified that setting $ls_{max} = 8$ and $lt_{max} = 8$ is enough for most translation tasks.

This algorithm shows a naive implementation of phrase extraction. Obviously, it can be improved in several ways, for example, [Koehn, 2010] describes a smarter algorithm to do the same thing with a lower time complexity. We refer readers to [Koehn, 2010] for more details and discussions on this issue.

It should also be noted that Koehn et al. [2003]'s model is not the only model in phrase-based MT. There are several variants of phrase-based model. For example, Marcu and Wong [2002] proposed a general form of phrase-based model where word alignment is not strictly required. Though these models and approaches are not supported in NiuTrans currently, they are worth an implementation in the future version of NiuTrans.

3.1.4 Reordering

Phrase reordering is a very important issue in current phrase-based models. Even if we know the correct translation of each individual phrase, we still need to search for a good reordering of them and generate a fluent translation. The first issue that arises is how to access all possible reordering efficiently. As arbitrary permutation of source phrases results in a extremely large number of reordering patterns (exponential in the number of phrases), the NiuTrans system restricts itself to a reordering model that is consistent with

Algorithm (straightforward implementation of phrase extraction)

Input: source string $\mathbf{s} = s_1...s_J$, target string $\mathbf{t} = t_1...t_I$ and word alignment matrix $\mathbf{a}$
Output: all phrase-pairs that are consistent with word alignments

```
1:  Function ExtractAllPhrases(s, t, a)
2:    for j1 = 1 to J                                  ▷ beginning of source phrase
3:      for j2 = j1 to j1 + ls_max − 1                 ▷ ending of source phrase
4:        for i1 = 1 to I                              ▷ beginning of target phrase
5:          for i2 = i1 to i1 + lt_max − 1             ▷ ending of target phrase
6:            if IsValid(j1, j2, i1, i2, a) then
7:              add phrase(j1, j2, i1, i2) into plist
8:    return plist
9:  Function IsValid(j1, j2, i1, i2, a)
10:   for j = j1 to j2
11:     if ∃i' ∉ (i1, i2) : a[j, i'] = 1 then        ▷ if a source word is aligned outside the target phrase
12:       return false;
13:   for i = i1 to i2
14:     if ∃j' ∉ (j1, j2) : a[j', i] = 1 then        ▷ if a target word is aligned outside the source phrase
15:       return false;
16:   return true;
```

Figure 3.2. Phrase Extraction Algorithm

Bracketing Transduction Grammars (BTGs). Generally speaking, BTG can be regarded as a special instance of Inversion Transduction Grammars (ITGs) [Wu, 1997]. Its major advantage is that all possible reorderings can be compactly represented with binary bracketing constraints. In the BTG framework, the generation from a source string to a target string is derived using only three types of rules:

$$X \rightarrow X_1\ X_2, \quad X_1\ X_2 \tag{R1}$$
$$X \rightarrow X_1\ X_2, \quad X_2\ X_1 \tag{R2}$$
$$X \rightarrow \bar{s}, \quad \bar{t} \tag{R3}$$

where X is the only non-terminal in BTG. Rule R1 indicates the monotonic translation which merges two blocks (or phrase-pairs) into a larger block in the straight order, while rule R2 merges them in the inverted order. They are used to model the reordering problem. Rule R3 indicates the translation of basic phrase (i.e., phrase translation problem), and is generally called the lexical translation rule.

With the use of the BTG constraint, NiuTrans chooses two state-of-the-art reordering models, including a ME-based lexicalized reordering model and a MSD reordering model.

ME-based Lexicalized Reordering Model:

The Maximum Entropy (ME)-based reordering model [Xiong et al., 2006] only works with BTG-based MT systems. This model directly models the reordering problem with the probability outputted by a binary classifier. Given two blocks X_1 and X_2, the reordering probability of (X_1, X_2) can be defined as:

$$f_{BTG} = \Pr(o|X_1, X_2) \tag{3.3}$$

where X_1 and X_2 are two adjacent blocks that need to be merged into a larger block. o is their order, which covers the value over *straight*, *inverted*. If they are merged using the straight rule (R1), $o = straight$. On the other hand, if they are merged using the inverted rule (R2), $o = inverted$. Obviously, this problem can be cast as a binary classification problem. That is, given two adjacent blocks (X_1, X_2), we need to decide whether they are merged in the straight way or not. Following Xiong et al. (2006)'s work, eight features are integrated into the model to predict the order of two blocks. See Figure 3.3 of an illustration of the features used in the model. All the features are combined in a log-linear way (as in the standard ME model) and optimized using the standard gradient descent algorithms such as GIS and L-BFGS.

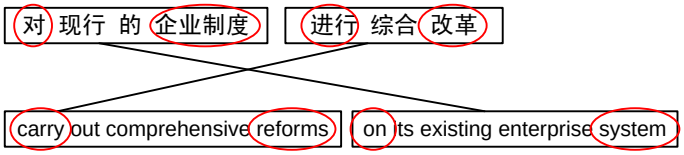


Figure 3.3. Example of the features used in the ME-based lexicalized reordering model. The red circles indicate the boundary words for defining the features.

For a derivation of phrase-pairs d[2], the score of ME-based reordering model in NiuTrans is defined to be:

$$f_{ME}(d) = \prod_{<o,X_1,X_2>\in d} \Pr(o|X_1, X_2) \tag{3.4}$$

where $f_{ME}(d)$ models the reordering of the entire derivation (using independent assumptions), and $\Pr(o|X_1, X_2)$ is the reordering probability of each pair of individual blocks.

MSD Reordering Model:

The second reordering model in NiuTrans is nearly the same as the MSD model used in [Tillman, 2004; Koehn et al., 2007; Galley and Manning, 2008]. For any phrase-pair, the MSD model defines three orientations with respect to the previous phrase-pair: monotone (M), swap (S), and discontinuous (D)[3]. Figure 3.4 shows an example of phrase orientations in target-to-source direction.

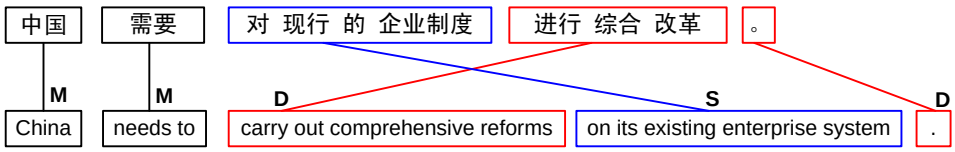
More formally, let $\mathbf{s} = \bar{s}_1...\bar{s}_K$ be a sequence of source-language phrases, and $\mathbf{t} = \bar{s}_1...\bar{t}_K$ be the sequence of corresponding target-language phrases, and $\mathbf{a} = \bar{a}_t...\bar{a}_K$ be the alignments between $\mathbf{s}$ and $\mathbf{t}$ where $\bar{s}_{a_i}$ is aligned with t_i. The MSD reordering score is defined by a product of probabilities of orientations $\mathbf{o} = \bar{o}_1...\bar{o}_K$.

$$\Pr(\mathbf{o}|\mathbf{s}, \mathbf{t}, \mathbf{a}) = \prod_{i=1}^{K} \Pr(o_i|\bar{s}_{a_i}, \bar{t}_i, a_{i-1}, a_i) \tag{3.5}$$

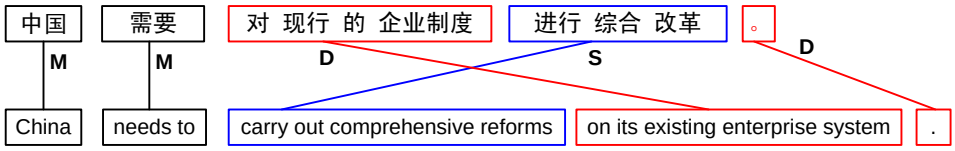
where o_i takes values over $O = \{M, S, D\}$ and is conditioned on a_{i-1} and a_i:

[2]The concept of *derivation* will be introduced in Section 3.1.7

[3]Note that the discontinuous orientation is actually no use for BTGs. In NiuTrans, it is only considered in the training stage and would not affect the decoding process.



(a) target-to-source



(b) source-to-target

Figure 3.4. Illustration of the MSD reoredering model. The phrase pairs with swap (S) and discontinuous (D) orientations are marked in blue and red, respectively. This model can handle the swap of a prepositional phrase "on its existing enterprise system" with a verb phrase "carry out comprehensive reforms".

$$o_i = \begin{cases} M & \text{if } a_i - a_{i-1} = 1 \\ S & \text{if } a_i - a_{i-1} = -1 \\ D & \text{otherwise} \end{cases}$$

Then, three feature functions are designed to model the reordering problem. Each corresponds an orientation.

$$f_{M-pre}(d) = \prod_{i=1}^{K} \Pr(o_i = M | \bar{s}_{a_i}, \bar{t}_i, a_{i-1}, a_i) \quad (3.6)$$

$$f_{S-pre}(d) = \prod_{i=1}^{K} \Pr(o_i = S | \bar{s}_{a_i}, \bar{t}_i, a_{i-1}, a_i) \quad (3.7)$$

$$f_{D-pre}(d) = \prod_{i=1}^{K} \Pr(o_i = D | \bar{s}_{a_i}, \bar{t}_i, a_{i-1}, a_i) \quad (3.8)$$

In addition to the three features described above, three similar features ($f_{M-fol}(d)$, $f_{S-fol}(d)$ and $f_{D-fol}(d)$) can be induced according to the orientations determined with respect to the following phrase-pair instead of the previous phrase-pair. i.e., o_i is conditioned on (a_i, a_{i+1}) instead of (a_{i-1}, a_i).

In the NiuTrans system, two approaches are used to estimate the probability $\Pr(o_i|\bar{s}_{a_i}, \bar{t}_i, a_{i-1}, a_i)$ (or $\Pr(o_i|\bar{s}_{a_i}, \bar{t}_i, a_i, a_{i+1})$). Supposing that $\bar{t}_i$ spans the word range $(t_u, ..., t_v)$ on the target-side, and $\bar{s}_{a_i}$ spans the word range $(s_x, ..., s_y)$ on the source side, $\Pr(o_i|\bar{s}_{a_i}, \bar{t}_i, a_{i-1}, a_i)$ can be computed in the following two ways:

- Word-based Orientation Model [Koehn et al., 2007]. This model checks the present of word alignments at $(x-1, u-1)$ and $(x-1, v+1)$. $o_i = M$ if $(x-1, u-1)$ has a word alignment. $o_i = S$ if $(x-1, u-1)$ does not have an alignment and $(x-1, v+1)$ has an alignment. Otherwise, $o_i = D$. Figure 3.5(a) shows an example of the "$o_i = S$" case. Once orientation o_i is determined, $\Pr(o_i|\bar{s}_{a_i}, \bar{t}_i, a_{i-1}, a_i)$ can be estimated from the training data by using relative frequency estimate.

- Phrase-based Orientation Model [Galley and Manning, 2008]. This model decides o_i based on adjacent phrases. $o_i = M$ if a phrase-pair can be extracted at $(x-1, u-1)$ given no constraint on maximum phrase length. $o_i = S$ if a phrase-pair can be extracted at $(x-1, v+1)$. Otherwise, $o_i = D$. Figure 3.5(b) shows an example of the "$o_i = S$" case in this model. Like the word-based counterpart, $\Pr(o_i|\bar{s}_{a_i}, \bar{t}_i, a_{i-1}, a_i)$ is also estimated by relative frequency estimate.

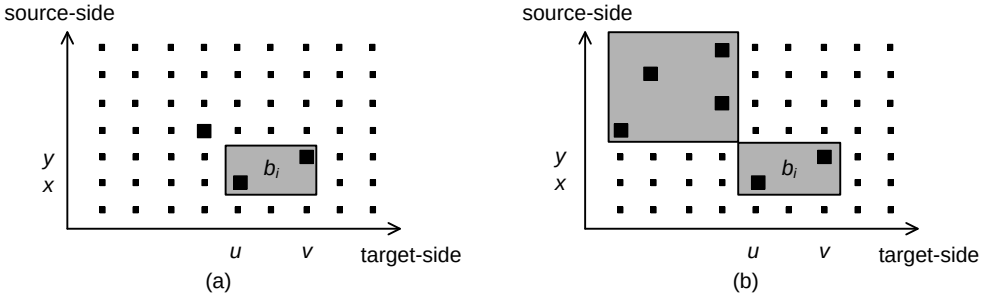


Figure 3.5. Examples of swap (S) orientation in the two models. $(\bar{s}_{a_i}, \bar{t}_i)$ is denoted as b_i (the i-th block). Black squares denote the present of word alignment, and grey rectangles denote the phrase pairs extracted without the constraint on phrase length. In (a), the orientation of b_i is recognized as swap (S) according to both models, while in (b) the orientation of b_i is recognized as swap (S) only by phrase-based orientation model.

It is trivial to integrate the above two reordering models into decoding. All you need is to calculate the corresponding (reordering) score when two hypotheses are composed. Please refer to Section 3.1.7 for more details about decoding with BTGs.

3.1.5 Features Used in NiuTrans.Phrase

A number of features are used in NiuTrans. Some of them are analogous to the feature set used other state-of-the-art systems such as Moses [Koehn et al., 2007]. The following is a summarization of the NiuTrans's feature set.

- **Phrase translation probability** $\Pr(\bar{t}|\bar{s})$. This feature is found to be helpful in most of previous phrase-based systems. It is obtained using maximum likelihood estimation (MLE).

$$\Pr(\bar{t}|\bar{s}) = \frac{count(\bar{t})}{count(\bar{s}, \bar{t})} \tag{3.9}$$

- **Inverted phrase translation probability** $\Pr(\bar{s}|\bar{t})$. Similar to $\Pr(\bar{t}|\bar{s})$, but with an inverted direction.

- **Lexical weight** $\Pr_{lex}(\bar{t}|\bar{s})$. This feature explains how well the words in $\bar{s}$ align the words in $\bar{t}$. Suppose that $\bar{s} = s_1...s_J$, $\bar{t} = t_1...t_I$ and $\mathbf{a}$ is the word alignment between $s_1...s_J$ and $t_1...t_I$. $\Pr_{lex}(\bar{t}|\bar{s})$ is calculated as follows:

$$\Pr_{lex}(\bar{t}|\bar{s}) = \prod_{j=1}^{J} \frac{1}{|\{j|a(j,i)=1\}|} \sum_{\forall(j,i):a(j,i)=1} w(t_i|s_j) \tag{3.10}$$

where $w(t_i|s_j)$ is the weight for (s_j, t_i).

- **Inverted lexical weight** $\Pr_{lex}(\bar{s}|\bar{t})$. Similar to $\Pr_{lex}(\bar{t}|\bar{s})$, but with an inverted direction.
- **N-gram language model** $\Pr_{lm}(\mathbf{t})$. A standard n-gram language model, as in other SMT systems.
- **Target word bonus (TWB)** $length(\mathbf{t})$. This feature is used to eliminate the bias of n-gram LM which prefers shorter translations.
- **Phrase bonus (PB)**. Given a derivation of phrase-pairs, this feature counts the number of phrase-pairs involved in the derivation. It allows the system to learn a preference for longer or shorter derivations.
- **Word deletion bonus (WDB)**. This feature counts the number of word deletions (or explicit null-translations) in a derivation. It allows the system to learn how often word deletion is performed.
- **ME-based reordering model** $f_{ME}(d)$. See Section 3.1.4.
- **MSD-based reordering model** $f_{M-pre}(d)$, $f_{S-pre}(d)$, $f_{D-pre}(d)$, $f_{M-fol}(d)$, $f_{S-fol}(d)$ and $f_{D-fol}($ $d)$. See Section 3.1.4.

As mentioned previously, all the features used in NiuTrans are combined in a log-linear fashion. Given a derivation d, the corresponding model score is calculated by the following equation.

$$\begin{aligned}\Pr(\mathbf{t}, d|\mathbf{s}) &= \prod_{(\bar{s},\bar{t})\in d} score(\bar{s},\bar{t}) \times \\ & f_{ME}(d)^{\lambda_{ME}} \times f_{MSD}(d)^{\lambda_{MSD}} \times \\ & \Pr_{lm}(\mathbf{t})^{\lambda_{lm}} \times \exp(\lambda_{TWB} \cdot length(\mathbf{t}))/Z(\mathbf{s}) \end{aligned} \quad (3.11)$$

where $Z(\mathbf{s})$ is the normalization factor[4], $f_{ME}(d)$ and $f_{MSD}(d)$ are the reordering model scores, and $\Pr_{lm}(\mathbf{t})$ is the n-gram language model score. $score(phrase)$ is the weight defined on each individual phrase-pair:

$$\begin{aligned} score(\bar{s},\bar{t}) &= \Pr(\bar{t}|\bar{s})^{\lambda_1} \times \Pr(\bar{s}|\bar{t})^{\lambda_2} \times \Pr_{lex}(\bar{t}|\bar{s})^{\lambda_3} \times \Pr_{lex}(\bar{s}|\bar{t})^{\lambda_4} \times \\ & \exp(\lambda_{PB}) \times \exp(\lambda_{WDB} \cdot \delta(\bar{s} \rightarrow null)) \end{aligned} \quad (3.12)$$

3.1.6 Minimum Error Rate Training

To optimize the feature weights, Minimum Error Rate Training (MERT), an optimization algorithm introduced by Och [2003], is selected as the base learning algorithm in NiuTrans. The basic idea of MERT is to search for the optimal weights by minimizing a given error metric on the training set, or in other words, maximizing a given translation quality metric. Let $S = \mathbf{s}_1...\mathbf{s}_m$ be m source sentences, $u(\lambda)$ be an SMT system, $T(u(\lambda)) = \mathbf{t}_1...\mathbf{t}_m$ be the translations produced by $u(\lambda)$, and $R = \mathbf{r}_1...\mathbf{r}_m$ be the reference translations where $\mathbf{r}_i = \{r_{i1}, ..., r_{iN}\}$. The objective of MERT can be defined as:

[4] $Z(\mathbf{s})$ is not really considered in the implementation since it is a constant with respect to $\mathbf{s}$ and does not affect the $argmax$ operation in Equation (3.1)

$$\lambda^* = \arg\min_{\lambda} \mathbf{Err}(T(u(\lambda)), R) \tag{3.13}$$

where **Err** is an error rate function. Generally, Err is defined with an automatic metric that measures the number of errors in $T(u(\lambda))$ with respect to the reference translations R. Since any evaluation criterion can be used to define **Err**, MERT can seek a tighter connection between the feature weights and the translation quality. However, involving MT evaluation metrics generally results in an unsmoothed error surface, which makes the straightforward solution of Equation (3.13) is not trivial. To address this issue, Och [2003] developed a grid-based line search algorithm (something like the Powell search algorithm) to approximately solve Equation (3.13) by performing a series of one-dimensional optimizations of the feature weight vector, even if **Err** is a discontinuous and non-differentiable function. While Och's method cannot guarantee to find the global optima, it has been recognized as a standard solution to learning feature weights for current SMT systems due to its simplicity and effectiveness.

Like most state-of-the-art SMT systems [Chiang, 2005; Koehn et al., 2007], BLEU is selected as the accuracy measure to define the error function used in MERT. In this way, the error rate function in NiuTrans is defined to be:

$$\mathbf{Err}(T(u(\lambda)), R) = 1 - \mathbf{BLEU}(T(u(\lambda)), R) \tag{3.14}$$

where $\mathbf{BLEU}(T(u(\lambda)), R)$ is the BLEU score of $T(u(\lambda))$ with respect to R.

3.1.7 Decoding

The goal of decoding is to search for the best translation for a given source sentence and trained model. As is introduced in Section 3.1.1, the posterior probability $\Pr(\mathbf{t}|\mathbf{s})$ is modeled on the input string and output string $(\mathbf{s}, \mathbf{t})$. But all the features designed above are associated with a derivation of phrase-pairs, rather than $(\mathbf{s}, \mathbf{t})$. Fortunately, the following rule can be used to compute $\Pr(\mathbf{t}|\mathbf{s})$ by summing over all derivations' probabilities.

$$\Pr(\mathbf{t}|\mathbf{s}) = \sum_{d \in D(\mathbf{s},\mathbf{t})} \Pr(\mathbf{t}, d|\mathbf{s}) \tag{3.15}$$

where $D(\mathbf{s}, \mathbf{t})$ is the derivation space for $(\mathbf{s}, \mathbf{t})$. Hence Equation (3.1) can be re-written as:

$$\mathbf{t}^* = \arg\max_{\mathbf{t}} \sum_{d \in D(\mathbf{s},\mathbf{t})} \Pr(\mathbf{t}, d|\mathbf{s}) \tag{3.16}$$

However, $D(\mathbf{s}, \mathbf{t})$ is generally a very large space. As a consequence it is inefficient (even impractical in most cases) to enumerate all derivations in $D(\mathbf{s}, \mathbf{t})$, especially when the n-gram language model is integrated into the decoding. To address this issue, a commonly-used solution is to use 1-best (Viterbi) derivation to represent the set of derivations for $(\mathbf{s}, \mathbf{t})$. In this way, the decoding problem can be formulized using the Viterbi decoding rule:

$$\mathbf{t}^* = \arg\max_{\mathbf{t}} \max_{d \in D(\mathbf{s},\mathbf{t})} \Pr(\mathbf{t}, d|\mathbf{s}) \tag{3.17}$$

As BTG is involved, the CKY algorithm is selected to solve the *argmax* operation in the above equation. In NiuTrans's decoder, each source span is associated with a data structure called *cell*. It records all the partial translation hypotheses (derivations) that can be mapped onto the span. Given a source sentence, all the cells are initialized with the phrase translations appear in the phrase table. Then, the decoder works in a bottom-up fashion, guaranteeing that all the sub-cells within $cell[j_1, j_2]$ are expended before $cell[j_1, j_2]$ is expended. The derivations in $cell[j_1, j_2]$ are generated by composing each pair of neighbor sub-cells within $cell[j_1, j_2]$ using the monotonic or inverted translation rule. Meanwhile the associated model score is calculated using the log-linear model described in Equation (3.11). Finally, the decoding completes when the entire span is reached. Figure 3.6 shows the pseudocode of the CKY-style decoding algorithm used in NiuTrans.

The CKY-style decoding Algorithm

Input: source string $\mathbf{s} = s_1...s_J$, and the model u with weights λ
Output: (1-best) translation

```
1:  Function CKYDecoding(s, u, λ)
2:    foreach (j1, j2): 1 ≤ j1 ≤ J and 1 ≤ j2 ≤ J
3:      initialize cell[j1, j2] with u and λ
4:    for j1 = 1 to J                                  ▷ beginning of span
5:      for j2 = j1 to J                               ▷ ending of span
6:        for k = j1 to j2                             ▷ partition of span
7:          hypos = Compose(cell[j1, k], cell[k, j2])
8:          cell[j1, j2[.update(hypos)
9:    return cell[1, J].1best()
10: Function Compose(cell[j1, k], cell[k, j2], u, λ)
11:   newhypos = ∅
12:     foreach hypo1 in cell[j1, k]                   ▷ for each hypothesis in the left span
13:       foreach hypo2 in cell[k, j2]                 ▷ for each hypothesis in the right span
14:         newhypos.add(straight(hypo1, hypo2))       ▷ straight translation
15:         newhypos.add(inverted(hypo1, hypo2))       ▷ inverted translation
16:   return newhypos
```

Figure 3.6. The CKY-style decoding algorithm

It is worth noting that a naive implementation of the above algorithm may result in very low decoding speed due to the extremely large search space. In NiuTrans, several pruning methods are used to speed-up the system, such as beam pruning and cube pruning. In this document we do not discuss more about these techniques. We refer readers to [Koehn, 2010] for a more detailed description of the pruning techniques.

3.1.8 Automatic Evaluation (BLEU)

Once decoding is finished, automatic evaluation is needed to measure the translation quality. Also, the development (or tuning) of SMT systems require some metrics to tell us how good/bad the system output

is. Like most related systems, NiuTrans chooses BLEU as the primary evaluation metric. As mentioned in Section 3.1.6, the BLEU metric can also be employed to define the error function used in MERT.

Here we give a brief introduction of BLEU. Given m source sentences, a sequence of translations $T = \mathbf{t}_1...\mathbf{t}_m$, and a sequence of reference translations $R = \mathbf{r}_1...\mathbf{r}_m$ where $\mathbf{r}_i = \{r_{i1}, ..., r_{iN}\}$, the BLEU score of T is defined to be:

$$\mathbf{BLEU}(T, R) = \mathbf{BP}(T, R) \times \prod_{n=1}^{4} \mathbf{Precision}_n(T, R) \tag{3.18}$$

where $\mathbf{BP}(T, R)$ is the brevity penalty and $\mathbf{Precision}_n(T, R)$ is the n-gram precision. To define these two factors, we follow the notations introduced in [Chiang et al., 2008] and use *multi-sets* in the following definitions. Let X be a multi-set, and $\#_X(a)$ be the number of times a appears in X. The following rules are used to define multi-sets:

$$|X| = \sum_a \#_X(a) \tag{3.19}$$

$$\#_{X \cap Y} = min(\#_X(a), \#_Y(a)) \tag{3.20}$$

$$\#_{X \cup Y} = max(\#_X(a), \#_Y(a)) \tag{3.21}$$

Then, let $g_n(w)$ be the multi-set of all n-grams in a string w. The n-gram precision is defined as:

$$\mathbf{Precision}_n(T, R) = \frac{\sum_{i=1}^{m} |g_n(\mathbf{t}_i) \bigcap (\bigcup_{j=1}^{N} g_n(r_{ij}))|}{\sum_{i=1}^{m} |g_n(\mathbf{t}_i)|} \tag{3.22}$$

where $\sum_{i=1}^{m} |g_n(\mathbf{t}_i)|$ counts the number of n-gram in MT output, and $\sum_{i=1}^{m} |g_n(\mathbf{t}_i) \bigcap (\bigcup_{j=1}^{N} g_n(r_{ij}))|$ counts the clipping presents of n-grams in the reference translations.

As n-gram precision prefers the translation with fewer words. $\mathbf{BP}(T, R)$ is introduced to penalize short translations. It has the following form:

$$\mathbf{BP}(T, R) = \exp\left(1 - \max\left\{1, \frac{l_R(T)}{\sum_{i=1}^{m} |g_1(\mathbf{t}_i)|}\right\}\right) \tag{3.23}$$

where $l_R(T)$ where is the *effective reference length* of R with respect to T. There are three choices to define $\mathbf{BP}(T, R)$ which in turn results in different versions of BLEU: NIST-version BLEU , IBM-version BLEU [Papineni et al., 2002] and BLEU-SBP [Chiang et al., 2008].

In the IBM-version BLEU, the effective reference length is defined to be the length of reference translation whose length is closest to t_i:

$$\mathbf{BP}_{IBM}(T, R) = \exp\left(1 - \max\left\{1, \frac{\sum_{i=1}^{m} |\arg\min_{r_{ij}}(|\mathbf{t}_i| - |r_{ij}|)|}{\sum_{i=1}^{m} |g_1(\mathbf{t}_i)|}\right\}\right) \tag{3.24}$$

In the NIST-version BLEU, the effective reference length is defined as the length of the shortest reference translation:

$$\mathbf{BP}_{NIST}(T,R) = \exp\left(1 - \max\left\{1, \frac{\sum_{i=1}^{m} min\{|r_{i1}|, ..., |r_{iN}|\}}{\sum_{i=1}^{m} |g_1(\mathbf{t}_i)|}\right\}\right) \tag{3.25}$$

BLEU-SBP uses a *strict brevity penalty* which clips the per-sentence reference length:

$$\mathbf{BP}_{SBP}(T,R) = \exp\left(1 - \max\left\{1, \frac{\sum_{i=1}^{m} |\arg\min_{r_{ij}}(|\mathbf{t}_i| - |r_{ij}|)|}{\sum_{i=1}^{m} min\{|\mathbf{t}_i|, |r_{i1}|, ..., |r_{iN}|\}}\right\}\right) \tag{3.26}$$

In NiuTrans, all three versions of the above BLEU scores are supposed. For example, users can choose which matric is used in MER training as needed.

3.2 Step 1 - Phrase Extraction and Parameter Estimation

Next, we show how to use the NiuTrans.Phrase engine and introduce the detailed instructions to set-up the system. We start with phrase extraction and parameter estimation which are two early-stage components of the training pipeline. In NiuTarns, they are implemented in a single program, namely NiuTrans.PhraseExtractor (in /bin/). Basically, NiuTrans.PhraseExtractor have four functions which corresponds to the four steps in phrase extraction and parameter estimation.

- **Step 1**: Extract (plain) phrase-pairs from word-aligned sentence-pairs
- **Step 2**: Extract lexical translations from word-aligned sentence-pairs (for calculating lexical weights. See Section 3.1.5).
- **Step 3**: Obtain the associated scores for each phrase-pair.
- **Step 4**: Filter the phrase table

3.2.1 Phrase Extraction

As described above, the first step is learning phrase translations from word-aligned bilingual corpus. To extract various phrase-pairs (for both source-to-target and target-to-source directions), the following command is used in NiuTrans:

Command

```
$ cd NiuTrans/bin/
$ mkdir ../work/extract/ -p
$ ./NiuTrans.PhraseExtractor --EXTP \
      -src      ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tgt      ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -aln      ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -out      ../work/extract/extract \
      -srclen   3 \
      -tgtlen   5 \
      -null     1
```

where the following options MUST be specified:

`--EXTP`, which specifies the working directory for the program.

`-src`, which specifies the source-language side of the training data (one sentence per line).

`-tgt`, which specifies the target-language side of the training data (one sentence per line).

`-aln`, which specifies the word alignments between the source and target sentences.

`-out`, which specifies result file of extracted phrases.

There are some other (optional) options which can activate more functions for phrase extraction.

`-srclen`, which specifies the maximum length of source-phrase (set to 3 by default).

`-tgtlen`, which specifies the maximum length of target-phrase (set to 3 by default).

`-null`, which indicates whether null-translations are explicitly modeled and extracted from bilingual corpus. If `-null 1`, null-translations are considered; if `-null 0`, they are not explicitly considered.

Output: two files "extract" and "extract.inv" are generated in "/NiuTrans/work/extract/".

Output (/NiuTrans/work/extract/)

```
- extract            ▷ "source → target" phrases
- extract.inv        ▷ "target → source" phrases
```

3.2.2 Obtaining Lexical Translations

As two lexical weights are involved in the NiuTrans system (See $\Pr_{lex}(\bar{t}|\bar{s})$ and $\Pr_{lex}(\bar{s}|\bar{t})$ in Section 3.1.5), lexical translations are required before parameter estimation. The following instructions show how to obtain lexical translation files (in both source-to-target and target-to-source directions) in the NiuTrans system:

Command

```
$ cd NiuTrans/bin/
$ mkdir ../work/lex/ -p
$ ./NiuTrans.PhraseExtractor --LEX \
      -src   ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tgt   ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -aln   ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -out   ../work/lex/lex
```

where

`--LEX`, which indicates that the program (NiuTrans.PhraseExtractor) works for extracting lexical translations.

`-src`, which specifies the source sentences of bilingual training corpus.

`-tgt`, which specifies the target sentences of bilingual training corpus.

`-aln`, which specifies word alignments between the source and target sentences.

`-out`, which specifies the prefix of output files (i.e., lexical translation files)

Also, there are some optional parameters, as follows:

`-temp`, which specifies the directory for sorting temporary files generated during the processing.

`-stem`, which specifies whether stemming is used. e.g., if `-stem` is specified, all the words are stemmed.

Output: two files "lex.s2d.sorted" and "lex.d2s.sorted" are generated in "/NiuTrans/work/lex/".

Output (/NiuTrans/work/lex/)

```
- lex.s2d.sorted        ▷ "source → target" lexical translation file
- lex.d2s.sorted        ▷ "target → source" lexical translation file
```

3.2.3 Generating Phrase Translation Table

The next step is the generation of phrase translation table which will then be used in the decoding step. Basically the phrase table is a collections of phrase-pairs with associated scores (or features). In NiuTrans, all the phrase-pairs are sorted in alphabetical order, which makes the system can efficiently loads/organizes the phrase table in a internal data structure. Each entry of the phrase table is made up several fields. To illustrate their meaning, Figure 3.7 shows a sample table.

In this example, each line is separated into five fields using " ||| ". The meaning of them are:

- The **first** field is the source side of phrase-pair.
- The **second** field is the target side of phrase-pair.

Phrase Translation Table

```
...
baogao renwei ||| report holds that ||| -2.62104 -5.81374 -0.916291 -2.8562 1 0 ||| 4 ||| 0-0 1-1 1-2
, beishang ||| , sadness ||| -1.94591 -3.6595 0 -3.70918 1 0 ||| 1 ||| 0-0 1-1
, beijing deng ||| , beijing , and other ||| 0 -7.98329 0 -3.84311 1 0 ||| 2 ||| 0-0 1-1 2-2 2-3 2-4
, beijing ji ||| , beijing and ||| -0.693147 -1.45853 -0.916291 -4.80337 1 0 ||| 2 ||| 0-0 1-1 2-2
...
```

Figure 3.7. Example of phrase translation table

- The **third** field is the set of features associated with the entry. The first four features are $\Pr(\bar{t}|\bar{s})$, $\Pr_{lex}(\bar{t}|\bar{s})$, $\Pr(\bar{s}|\bar{t})$ and $\Pr_{lex}(\bar{s}|\bar{t})$ (See Section 3.1.5). The 5th feature is the phrase bonus exp(1). The 6th is an "undefined" feature which is reserved for feature engineering and can be defined by users.
- The **fourth** field is the frequency the phrase-pair appears in the extracted rule set. By using a predefined threshold (0), the phrase-pairs with a low frequency can be thrown away to reduce the table size and speed-up the system.
- The **fifth** field is the word alignment information. For example, in the first entry in Figure 3.7, word alignment "0-0 1-1 1-2" means that the first source word is aligned with the first target word, and the second source word is aligned with the second and third target words.

Then, the following instructions can be adopted to generate the phrase table from the extracted (plain) phrases and lexical translation tables:

Command

```
$ cd NiuTrans/bin/
$ mkdir ../work/model.tmp/ -p
$ ./NiuTrans.PhraseExtractor --SCORE \
      -tab      ../work/extract/extract \
      -tabinv   ../work/extract/extract.inv \
      -ls2d     ../work/lex/lex.s2d.sorted \
      -ld2s     ../work/lex/lex.d2s.sorted \
      -out      ../work/model.tmp/phrase.translation.table.step1
```

where

`--SCORE` indicates that the program (NiuTrans.PhraseExtractor) runs in the "scoring" mode. It scores each phrase-pair, removes the replicated entries, and sort the table.

`-tab` specifies the file of extracted rules in "source → target" direction.

`-tabinv` specifies the file of extracted rules in "target → source" direction.

`-ls2d` specifies the lexical translation table in "source → target" direction.

`-ld2s` specifies the lexical translation table in "target → source" direction.

`-out` specifies the resulting phrase table.

The optional parameters are:

`-cutoffInit` specifies the threshold for cutting off low-frequency phrase-pairs. e.g., "`-cutoffInit` = 1" means that the program would ignore the phrase-pairs that appear only once, while "`-cutoffInit` = 0" means that no phrases are discarded.

`-printAlign` specifies whether the alignment information (the 5th field) is outputted.

`-printFreq` specifies whether the frequency information (the 4th field) is outputted.

`-temp` specifies the directory for sorting temporary files generated in the above procedure.

Output: in this step four files are generated under "/NiuTrans/work/model.tmp/"

Output (/NiuTrans/work/model.tmp/)

```
- phrase.translation.table.step1                  ▷ phrase table
- phrase.translation.table.step1.inv              ▷ tmp file for rule extraction
- phrase.translation.table.step1.half.sorted      ▷ another tmp file
- phrase.translation.table.step1.half.inv.sorted  ▷ also a tmp file
```

Note that, "phrase.translation.table.step1" is the "real" phrase table which will be used in the following steps.

3.2.4 Table Filtering

As the phrase table contains all the phrase-pairs that extracted from the bilingual data, it generally suffers from its huge size. In some cases, even 100K bilingual sentences could result in tens of millions of extracted phrase-pairs. Obviously using/orgnizing such a large number of phrase-pairs burdens the system very much, even results in acceptable memory cost when a large training data-set is involved. A simple solution to this issue is filtering the table with the translations of test (and dev) sentences. In this method, we discard all the phrases containing the source words that are absent in the vocabulary extracted from test (or dev) sentences. Previous work has proved that such a method is very effective to reduce the size of phrase table. e.g., there is generally a 80% reduction when a relatively small set of test (or dev) sentences (less then 2K sentences) is used. It is worth noting that this method assumes an "off-line" translation environment, and is not applicable to online translation. In addition, another popular method for solving this issue is to limit the number of translation options for each source-phrase. This method is motivated by a fact that the low-probability phrase-pairs are seldom used during decoding. Thus we can rank the translation options with their associated probabilities (model score or $\Pr(\bar{t}|\bar{s})$) and keep the top-k options only. This way provides a flexible way to decide how big the table is and can work for both "off-line" and "on-line" translation tasks.

In NiuTrans, the maximum number of translation options (according to $\Pr(\bar{t}|\bar{s})$) can be set by users (See following instructions). The current version of the NiuTrans system does not support the filtering with test (or dev) sentences.

Command

```
$ cd NiuTrans/bin/
$ ./NiuTrans.PhraseExtractor --FILTN \
      -in       ../work/model.tmp/phrase.translation.table.step1 \
      -out      ../work/model.tmp/phrase.translation.table \
      -strict   30
```

where

`--FILTN` indicates that we run the program (NiuTrans.PhraseExtractor) to filtering the table.

`-in` specifies the input file (i.e., the phrase table)

`-out` specifies the output file (i.e., the filtered phrase table)

`-strict` specifies the maximum number of translation options for each source-phrase (30 by default).

Output: the filtered table ("phrase.translation.table") is placed in "NiuTrans/work/model.tmp/". It will be used as a sample phrase-table in the following illustration in this section.

Output (/NiuTrans/work/model.tmp/)

```
- phrase.translation.table  ▷ (filtered) phrase table for the following steps
```

3.3 Step 2 - Training Reordering Model

The following shows how to build the reordering models in NiuTrans.

3.3.1 ME-based Lexicalized Reordering Model

The NiuTrans system divides the training of ME-based lexicalized reordering model into three steps:

1. obtain the training samples (i.e., positive samples (straight translations) and negative samples (inverted translations)
2. train the ME classifier using the collected training samples.
3. transform the model file into the one used in NiuTrans.

These functions are implemented in three programs "NiuTrans.MEReorder", "maxent" and "dmconv.pl" which are all placed in the directory "NiuTrans/bin/". The following is a usage of them.

3.3.1.1 Obtaining Training Samples

First, the training samples are generated by the following instructions.

Command

```
$ cd NiuTrans/bin/
$ mkdir ../work/model.tmp/ -p
$ ./NiuTrans.MEReorder \
      -src              ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tgt              ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -algn             ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -output           ../work/model.tmp/me.reordering.table.step1 \
      -maxSrcPhrWdNum   3 \
      -maxTgtPhrWdNum   5 \
      -maxTgtGapWdNum   1 \
      -maxSampleNum     5000000
```

where

`-src`, `-tgt`, `-algn` specify the files of source sentences, target sentences and alignments between them, respectively.

`-out` specifies the output file

There are some other options which provide more useful functions for advanced users.

`-maxSrcPhrWdNum` specifies the maximum number of words in source spans (phrases) considered in training the model.

`-maxTgtPhrWdNum` specifies the maximum number of words in target spans (phrases) considered in training the model.

`-maxTgtGapWdNum` specifies the maximum number of unaligned words between the two target spans considered in training the model (further explanation is required!!! Illustration!!!).

`-maxSampleNum` specifies the maximum number of training samples generated for training the ME model. Since a large number of training samples would result in a very low speed of ME training, it is reasonable to control the number of training samples and generate a "small" model. The parameter `-maxSampleNum` offers a way to do this job.

Output: the resulting file is named as "me.reordering.table" and placed in "NiuTrans/work/model.tmp/".

Output (/NiuTrans/work/model.tmp/)

```
- me.reordering.table.step1    ▷ training samples for the ME-based model
```

3.3.1.2 Training the ME model

Then the ME model is learned by using the following commands:

Command

```
$ cd NiuTrans/bin/
$ ./maxent
      -i        200 \
      -g        1 \
      -m        ../work/model.tmp/me.reordering.table.step2 \
                ../work/model.tmp/me.reordering.table.step1 \
      --lbfgs
```

where

`-i` specifies the iteration number for training.

`-g` specifies the gaussian prior used in smoothing the parameters.

`-m` specifies the resulting model file.

`--lbfgs` indicates that the optimization method is L-BFGS.

Output: the model file "me.reordering.table.step2" is generated in "NiuTrans/work/model.tmp/".

Output (/NiuTrans/work/model.tmp/)

```
- me.reordering.table.step2    ▷ the model learned using ME
```

3.3.1.3 Generating the Model File Used in NiuTrans

Last, "me.reordering.table.step2" is transformed into the file used in NiuTrans by the script "dm-conv.pl".

Command

```
$ cd NiuTrans/scripts/
$ perl dm-conv.pl \
           ../work/model.tmp/me.reordering.table.step2 \
      >    ../work/model.tmp/me.reordering.table
```

The output is the file "me.reordering.table" in "NiuTrans/work/model.tmp/". This file can be used in the following decoding steps.

Output (/NiuTrans/work/model.tmp/)

```
- me.reordering.table        ▷ the ME-based reordering model
```

3.3.2 MSD Reodering Model

To learn the MSD reordering model, two steps are required in NiuTrans.

1. Obtain the MSD model from the word-aligned bilingual sentences.
2. Filter the model file with the phrase table extracted in the previous steps.

The purpose of the second step is to discard the entries that are absent in the filtered phrase table and thus reduce the model size.

3.3.2.1 Obtaining the Initial Model

To generate the initial model, please follow the following instructions:

Command

```
$ cd NiuTrans/bin/
$ ./NiuTrans.MSDReorder \
      -f    ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -e    ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -a    ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -m    1 \
      -o    ../work/model.tmp/msd.reordering.table.step1
$ rm smt_tmp_phrase_table*
```

where

`-f` specifies the file of source sentences.

`-e` specifies the file of target sentences.

`-a` specifies the file of word alignments.

`-o` specifies the output file

By default the MSD model is built using the word-based approach, as described in Section 3.1.4. Of course, users can use other variants as needed. Two optional parameters are provided within NiuTrans:

`-m` specifies the training method, where "1" indicates the word-based method and "2" indicates the "phrase-based" method. Its default value is "1".

`-max-phrase-len` specifies the maximum length of phrases (either source phrases or target phrases) considered in training. Its default value is +`infinite`.

Output: the resulting file is named as "msd.reordering.table.step1" and placed in "NiuTrans/work/model.tmp/".

Output (/NiuTrans/work/model.tmp/)

```
- msd.reordering.table.step1        ▷ the MSD reordering model
```

3.3.2.2 Filtering the MSD model

The MSD model (i.e., file "msd.reordering.table.step1") is then filtered with the phrase table, as follows:

Command

```
$ cd NiuTrans/scripts/
$ perl filter.msd.model.pl \
           ../work/model.tmp/phrase.translation.table \
           ../work/model.tmp/msd.reordering.table.step1 \
      >    ../work/model.tmp/msd.reordering.table
```

where "../work/model.tmp/msd.reordering.table" is the final MSD reordering model. Note that, model filtering is not a necessary step in NiuTrans. If sufficient memory resource is available and users do not care about the running-time (in the decoding stage) very much, it is suggested to straightforwardly use the model generated in Section 3.3.2.1 and skip this step[5].

3.4 Step 3 - N-gram Language Modeling

The NiuTrans package offers a n-gram language modeling tool (NiuTrans.LMTrainer). This tool is placed in "NiuTrans/bin/". To train the n-gram language model, users can repeat the instructions described in Section 2.2.

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/lm/ -p
$ perl NiuTrans-training-ngram-LM.pl \
      -corpus    ../sample-data/sample-submission-version/LM-training-set/e.lm.txt \
      -ngram     3 \
      -vocab     ../work/lm/lm.vocab \
      -lmbin     ../work/lm/lm.trie.data
```

where

`-corpus` specifies the training data file (i.e., a collection of target-language sentences. one sentence per line).

`-ngram` specifies the order of n-gram language model.

[5] You can rename the file "msd.reordering.table.step1" as "msd.reordering.table" directly

`-vocab` specifies the resulting vocabulary file.

`-lmbin` specifies the resulting model file.

In the above example, two files (vocabulary and model file) are generated under "/NiuTrans/work/lm/". They will be used the following steps of decoding.

Output (/NiuTrans/work/lm/)

```
- lm.vocab        ▷ vocabulary file
- lm.trie.data    ▷ model file of n-gram language model
```

3.5 Step 4 - Configuring the Decoder

3.5.1 Config file

Decoder is one of the most complicated components in modern SMT systems. Generally, many techniques (or tricks) are employed to successfully translate source sentences into target sentences. The NiuTrans system provides an easy way to setup the decoder using a config file. Hence users can choose different settings by modifying this file and setup their decoders for different tasks. NiuTrans' config file follows the "key-value" definition. The following is a sample file which offers most necessary settings of the NiuTrans.Phrase system[6].

The meanings of these parameters are:

- `Ngram-LanguageModel-File` specifies the n-gram language model file.
- `Target-Vocab-File` specifies the target-language vocabulary.
- `ME-Reordering-Table` specifies the ME-based lexicalized reordering model file.
- `MSD-Reordering-Model` specifies the MSD reordering model file.
- `Phrase-Table` specifies the phrase table.
- `nround` specifies how many rounds MERT performs. In each round of MERT run, the system produces the k-best translations and optimizes the feature weights.
- `ngram` specifies the order of n-gram language model used in decoding.
- `usepuncpruning` specifies whether the Punctuation Pruning is used (1: use punctuation pruning; 0: do not use it). If `usepuncpruning` is fired, the system would first divide the input sentence into smaller fragments according to punctuations (such as common). Then it decodes each fragment individually and glue their translations to generate the translation for the entire sentence.

[6] See "/config/NiuTrans.phrase.config" for a complete version of the config file

Decoder Config File (NiuTrans.Phrase)

```
param="Ngram-LanguageModel-File"    value="../sample-data/lm.trie.data"
param="Target-Vocab-File"           value="../sample-data/lm.vocab"
param="ME-Reordering-Table"         value="../sample-data/me.reordering.table"
param="MSD-Reordering-Model"        value="../sample-data/msd.reordering.table"
param="Phrase-Table"                value="../sample-data/phrase.translation.table"
param="nround"                      value="15"
param="ngram"                       value="3"
param="usepuncpruning"              value="1"
param="usecubepruning"              value="1"
param="use-me-reorder"              value="1"
param="use-msd-reorder"             value="1"
param="nthread"                     value="4"
param="nbest"                       value="30"
param="outputnull"                  value="0"
param="beamsize"                    value="30"
param="nref"                        value="1"
param="usenulltrans"                value="0"
param="normalizeoutput"             value="0"
param="weights"                     value="1.000 0.500 0.200 0.200 0.200 0.200 \
                                           0.500 0.500 -0.100 1.000 0.000 0.100 \
                                           0.100 0.100 0.100 0.100 0.100"
param="ranges"                      value="-3:7 -3:3 0:3 0:0.4 0:3 0:0.4 \
                                           -3:3 -3:3 -3:0 -3:3 0:0 0:3 \
                                           0:0.3 0:0.3 0:3 0:0.3 0:0.3"
param="fixedfs"                     value="0 0 0 0 0 0 0 0 0 \
                                           0 0 0 0 0 0 0 0 0"
```

Figure 3.8. Decoder Config File (NiuTrans.Phrase)

- `usecubepruning` specifies whether the Cube Pruning is used (1: use cube pruning; 0: do not use it). For more details about cube pruning, please refer to [Huang and Chiang, 2005].

- `use-me-reorder` specifies whether the ME-based lexicalized reordering model is used (1: use cube the model; 0: do not use it).

- `use-msd-reorder` specifies whether the MSD reordering model is used (1: use cube the model; 0: do not use it).

- `nthread` specifies the number of threads used in decoding source sentences. More threads means a higher speed. But, as most multi-thread programs, the speed improvement is very modest when a large number threads are activated. It is suggested to set `nthread` to $4 \sim 8$ on normal PC servers.

- `nbest` specifies the size of n-best list generated by the decoder. The direct use of n-best output is MERT which optimizes feature weights by promoting the "best-BLEU" candidate from n-best outputs of MT systems. Generally a large n-best list could result in more stable convergence of

MERT. However, a too large n-best does not really help.

- `outputnull` specifies whether OOV words and deleted words (null-translations) are outputted in final translations. When `outputnull` is fired, all those OOV or deleted words will be marked as "<something>". E.g., translation "I had a $< XX >$ day today!" indicates that XX is an OOV word or null-translation word that is deleted during decoding.

- `beamsize` specifies the size (or width) of beam used in beam search. A large beam could reduce the number of search errors, but in turn slows down the system.

- `nref` specifies how many reference translations are provided for MER training.

- `usenulltrans` specifies whether explicit word deletion is allowed in decoding. If `usenulltrans = 1`, the decoder would delete some source words. Note that this feature is also called "devil feature" since it hurts the performance in some cases. e.g., in most applications, users do not expect to delete content words. However, this feature does not consider such a factor. So please be careful when using this feature.

- `normalizeoutput` specifies whether the output is normalized. In NiuTrans, normalization is implemented according to the similar function used in `mteval-v13a.pl`[7].

- `weights` specifies the feature weights. In MERT, `weights` means the initial weights.

- `ranges` specifies the range (min and max values) for each individual feature during weight tuning. e.g., in the above example, the range of the first feature is "-3:7" which means that the compounding feature can only choose values over $[-3, 7]$.

- `fixedfs` specifies whether a feature weight is fixed (or not tuned) during MERT. "1" means the corresponding feature weight is fixed and not adjusted in weight tuning.

In NiuTrans the features are ordered as follows (See fields `weights`, `ranges` and `fixedfs`)

[7] ftp://jaguar.ncsl.nist.gov/mt/resources/mteval-v13a.pl

id	feature	initial-value	min-value	max-value
1	n-gram language model	1.000	-3.000	7.000
2	target word bonus	0.500	-3.000	3.000
3	phrase translation probability	0.200	0.000	3.000
4	lexical weight	0.200	0.000	0.400
5	inverted phrase translation probability	0.200	0.000	3.000
6	inverted lexical weight	0.200	0.000	0.400
7	phrase bonus	0.500	-3.000	3.000
8	user-defined feature (for future extension)	0.500	0.000	3.000
9	number of word-deletion operations	-0.100	-3.000	0.000
10	ME-based lexicalized reordering model	1.000	-3.000	3.000
11	undefined	0.000	0.000	0.000
12	MSD reordering model: Previous & Monotonic	0.100	0.000	3.000
13	MSD reordering model: Previous & Swap	0.100	0.000	0.300
14	MSD reordering model: Previous & Discontinuous	0.100	0.000	0.300
15	MSD reordering model: Following & Monotonic	0.100	0.000	3.000
16	MSD reordering model: Following & Swap	0.100	0.000	0.300
17	MSD reordering model: Following & Discontinuous	0.100	0.000	0.300

3.5.2 Generating the Config file

The training steps produce three tables "phrase.translation.table", "me.reordering.table" and "msd.reordering.table", as well as the language model files in "/work/model/". All these recourse files are used to generate the config file for decoding. The following script could generate the config file automatically (from a template file "config/NiuTrans.phrase.config").

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/model/ -p
$ mv    ../work/model.tmp/phrase.translation.table \
        ../work/model.tmp/msd.reordering.table \
        ../work/model.tmp/me.reordering.table \
        ../work/model
$ mkdir ../work/config/ -p
$ perl NiuTrans-phrase-generate-mert-config.pl \
      -tmdir    ../work/model/ \
      -lmdir    ../work/lm/ \
      -o        ../work/config/NiuTrans.phrase.user.config
```

The parameters of "NiuTrans-phrase-generate-mert-config.pl" are

`-tmdir` specifies the directory that keeps all the tables such as "phrase.translation.table".

`-lmdir` specifies the directory that keeps all the LM files such as "lm.trie.data" and "lm.vocab".

`-o` specifies the output file (i.e., a config file).

Output: The output is file "NiuTrans.phrase.user.config" in "NiuTrans/work/config/". Users can modify "NiuTrans.phrase.user.config" as needed.

Output (/NiuTrans/work/config/)

```
- NiuTrans.phrase.user.config        ▷ the config file for NiuTrans.Phrase
```

3.6 Step 5 - Weight Tuning

As the config file is used to control the decoding and weight tuning processes, running MERT is very trivial in *NiuTrans*. You can execute the following script to carry out the MER training.

Command

```
$ cd NiuTrans/scripts/
$ perl NiuTrans-phrase-mert-model.pl \
      -c          ../work/config/NiuTrans.phrase.user.config \
      -dev        ../sample-data/sample-submission-version/Dev-set/Niu.dev.txt \
      -nref       1 \
      -r          3 \
      -nthread    12 \
      -l          ../work/mert-model.log
```

where

`-c` specifies the config file

`-dev` specifies the development (or tuning) set used in MERT.

`-method` specifies the method for choosing the optimal feature weights over a sequence of MERT runs. if "`-method arg`" is used, the resulting weights are the average numbers of those MERT runs; if "`-method max`" is used, the max-BLEU weights are chosen. By default, `-method` is set to `avg`.

`-r` specifies the number of reference translations provided (in the dev-set).

`-nthread` specifies the number of threads used in running the decoder.

`-l` specifies the log file. By default, the system generates a file "mert-model.log" under the working directory.

After MER training, the optimized feature weights are automatically recorded in "NiuTrans/work/-config/NiuTrans.phrase.user.config" (last line). Then, the config file can be used when decoding new sentences.

3.7 Step 6 - Decoding

Last, users can decode new sentences with the trained model and optimized feature features[8]. The following instructions can be used:

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/trans.result/ -p
$ perl NiuTrans-phrase-decoder-model.pl \
      -c        ../work/config/NiuTrans.phrase.user.config \
      -test     ../sample-data/sample-submission-version/Test-set/Niu.test.txt \
      -output   ../work/trans.result/1best.out
```

where

`-c` specifies the config file (with optimized feature weights)

`-test` specifies the test set (one sentence per line).

`-output` specifies the file of translations.

Output: The (1-best) translation file "1best.out" in "/NiuTrans/work/trans.result/".

Output (/NiuTrans/work/trans.result/)

```
- 1best.out        ▷ translation result
```

[8]Users can modify "NiuTrans.phrase.user.config" by themselves before testing

Chapter 4

NiuTrans.Hierarchy/NiuTrans.Syntax - A Syntax-based Translation Engine

The NiuTrans package also includes translation engines based on hierarchical phrase-based and syntax-based models, namely *NiuTrans.Hierarchy* and *NiuTrans.Syntax*. Unlike phrase-based model, hierarchical phrase-based and syntax-based models implicitly/explicitly characterize the movement of hierarchical structures by linguistic notions of syntax, and thus are more powerful in dealing with long distance dependencies. Depending what type of syntax is used, different approaches can be used for building translation system. For example, when syntactic parsers are not available, NiuTrans.Hierarchy is no-doubt a good choice since it does not require the use of linguistically annotated corpus but still benefit from (informal) linguistically-motivated grammars. On the other hand, for languages with promising parsing accuracy, NiuTrans.Syntax is a nice solution to make use of the syntax on both(either) source and(or) target language side(s).

As argued in the NLP community, whether syntax is really helpful to MT is still a somewhat controversial issue. For example, for Chinese-English translation, syntax-based systems have shown very promising results, even achieve state-of-the-art performance on recent MT evaluation tasks, such as the NIST MT track. However, for European languages, we cannot draw similar conclusions as those discovered in Chinese-English translation, and the syntax-based systems still underperform the phrase-based counterparts. While the effectiveness of syntactic information on improving MT has not yet been fully examined, we believe that the use of syntax is a very promising direction in MT. This motivates the development of NiuTrans.Hierarchy/NiuTrans.Syntax. The goal of NiuTrans.Hierarchy/NiuTrans.Syntax is two fold. First, it offers fundamental components for building various (baseline) syntax-based systems. Second, it offers tools for exploring solutions toward the better use of syntax for machine translation.

To get the basic knowledge of NiuTrans.Hierarchy/NiuTrans.Syntax, please go through the remaining parts of this section. We will first review the background knowledge of hierarchical phrase-based and syntax-based MT, and then describe the usage of NiuTrans.Hierarchy/NiuTrans.Syntax in detail. Again, for guys who are familiar with MT, it is suggested to jump to Section 4.2 directly.

4.1 Background

4.1.1 Basic Concepts

The hierarchical phrase-based and syntax-based models follow the widely-adopted framework of SMT where ones need to carry out *training* and *decoding* to obtain final translations. Unlike phrase-based translation, both the hierarchical phrase-based and the syntax-based approaches model the translation process using a derivation of *translation rules* instead phrase-pairs. In NiuTrans, translation rules are defined based on Synchronous Context-Free/Tree-Substitution Grammars (SCFGs/STSGs). In the framework of SCFG/STSG, the translation model is encoded by a set of translation rules and obtained on (auto-parsed) bilingual corpus using various methods of rule extraction and parameters estimation [1]. By using the learned model, the system can decode new input sentences. Figure 4.1 shows the pipeline of building a hierarchical phrase-based system or a syntax-based system.

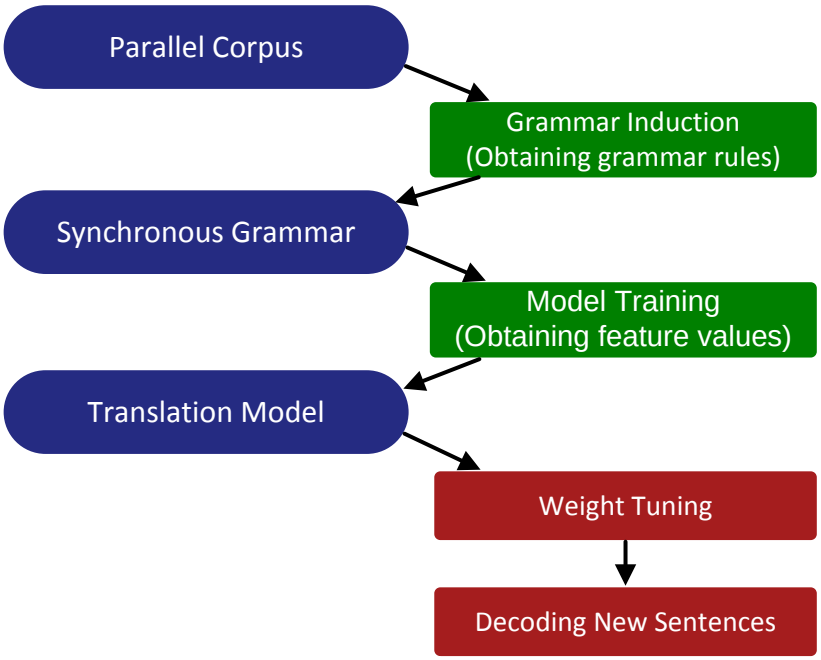


Figure 4.1. Pipeline of building (syntax-based) translation systems

In the following, many "new" concepts will be introduced. Most of them have not been mentioned in the previous sections. You may read the similar terms in MT-related papers. However, syntax-based MT is such an research area where many models have been developed but no agreed-framework is reached. Ones may read/hear different terminologies in different papers/talks which actually talk the same thing. To avoid the confusion about the terminologies used here, we list the common terms that appear in this document. Note that, our definition is just for clear presentation, rather than establishing the "correct" use of those terminologies (Actually some of them are still in question).

- **Translation rule** - atomic translation unit that the model operates on

[1] For the hierarchical phrase-based model, no parsed data is actually required.

- **Derivation** - a series of translation rules used to form the translation from a source tree/string to a target tree/string. It represents the mapping from a underlying source (syntactic) structure to a target (syntactic) structure according to the translation model
- **Rule set/table** - a set of translation rules as well as associated features
- **Hierarchical phrase-based** - model/approach that uses no linguistic syntax
- **String-to-tree** - model/approach that uses linguistic syntax on target-language side only
- **Tree-to-string** - model/approach that uses linguistic syntax on source-language side only
- **Tree-to-tree** - model/approach that uses linguistic syntax on both language sides
- **Syntax-based** - model/approach that uses linguistic syntax on both/either language sides/side
- **Tree-based** - model/approach that uses syntactic trees[2] as input when translates new sentences
- **String-based** - model/approach that uses string as input when translates new sentences

4.1.2 Synchronous Context-Free/Tree-Substitution Grammar

As stated in related studies, phrase-based models have a number of drawbacks though widely used. For example, the movement of hierarchical structures cannot be described within such type of model. Actually, the behaviors of moving hierarchical structures are more likely to be represented by linguistic notions of syntax, as what human translators may imagine during translation. Thus the syntax of languages can provide many hints and advantages to modeling this process.

4.1.2.1 SCFG

To date, there have been a number of attempts to apply syntactic knowledge to SMT. Some of them have confirmed the success of syntax-based MT, although there are plenty of exceptions. In NiuTrans, we choose Synchronous Context-Free Grammars (SCFGs) as the basis of the hierarchical phrase-based and syntax-based engines. SCFG is a natural generalization of Context-Free Grammar (CFG), and fit for parsing with both languages in translation. Numerous different approaches to SMT can be expressed in the framework of SCFG, such as syntax-directed translation [Aho and Ullman, 1969], inversion transduction grammar [Wu, 1997] and head transducers [Alshawi et al., 2000]. The main advantage of applying SCFG to MT is that many theories and techniques in CFG parsing are directly applicable to translation with SCFG. Here we give a brief introduction to SCFG to ease the understanding of our presentation. For a finer-grained description, please refer to [Chiang and Knight, 2006].

In the formalism of SCFG, the translation problem can be cast as transformation from an input string to a target string. This procedure is essentially a parsing problem, where the atomic step is the mapping of terminal/non-terminal sequences between two languages. In SCFG it is realized by the so-called *grammar rules* which can be regarded as a generalization of CFG rules to the case of two output strings. Recall that a

[2]Here we refer syntactic tree as phrase structure tree

CFG consists of a terminal symbol set T, a non-terminal symbol set N, and a rule set $R = \{N \rightarrow N^* \bigcup T^*\}$. When parsing a sentence, we start with a start symbol, and then recursively rewrite the non-terminal symbols with CFG rules until no non-terminals are left. The output is a derivation of rule applications that forms the tree structure rooting at the start symbol and yielding the input sentence. The following shows some CFG rules induced from the English Penn Treebank, where the non-terminal symbols represent syntactic categories and the terminals represent words.

$$\text{S} \longrightarrow \text{NP VP} \quad \text{(C1)}$$
$$\text{NP} \longrightarrow \text{DT NN} \quad \text{(C2)}$$
$$\text{VP} \longrightarrow \text{VBZ} \quad \text{(C3)}$$
$$\text{DT} \longrightarrow \text{the} \quad \text{(C4)}$$
$$\text{NN} \longrightarrow \text{boy} \quad \text{(C5)}$$
$$\text{VBZ} \longrightarrow \text{falls} \quad \text{(C6)}$$

In CFG, the left-hand side of rule is the root symbol of the production, and the right-hand side is the output string. Unlike CFG, SCFG rule has two right-hand sides (i.e., two output strings). One is for source language output and the other is for target language output. Many translation equivalence models can be expressed by the formulism of SCFG. Here we choose the hierarchical phrase-based model as an instance to give an intuitive explanation of SCFG. Let us see some rules in a sample SCFG first.

$$\text{X} \longrightarrow \text{jinkou , the imports} \quad \text{(S1)}$$
$$\text{X} \longrightarrow \text{jianshao , fall} \quad \text{(S2)}$$
$$\text{X} \longrightarrow \text{X}_1 \text{ dafudu } \text{X}_2 \text{ , } \text{X}_1 \text{ drastically } \text{X}_2 \quad \text{(S3)}$$

Here the two right-hand sides are separated by ",". The subscripts indicates the one-to-one mappings between the non-terminals of the two right-hand sides. E.g., source-language X_1 links target-language X_1, source-language X_2 links target-language X_2 and so on. In the hierarchical phrase-based model, there is only one non-terminal X, and the output string is a sequence of terminals and non-terminals. As in CFG paring, the non-terminal can be viewed as a variable and need to be replaced by other rules during the parsing process. The replacement of variable is generally called the *rewritten* operation. In the SCFG model, given a source sentence and target sentence, we start with a pair of start symbols and repeatedly rewrite pairs of nonterminal symbols using the SCFG rules, with the constraint that the labels of the rewritten non-terminals must match the root labels of the rewriting rules. See Figure 4.2 for an illustration of the steps used in parsing a sample sentence pair with rules S1-S3.

The hierarchical phrase-based model is generally regarded as the simplest instance of the general framework of SCFG. In this model, the non-terminals do not have linguistic meanings, and the grammar represents an informal syntax of recursive structure of language. Of course, it can be enhanced by introducing the notations of the real syntax used in language parsing. Suppose that we are in Chinese-to-English translation. We can annotate the non-terminals in rules S1-S3 with the labels defined in the English Penn Treebank. Then we obtain the following rules annotated with the target-language syntax.

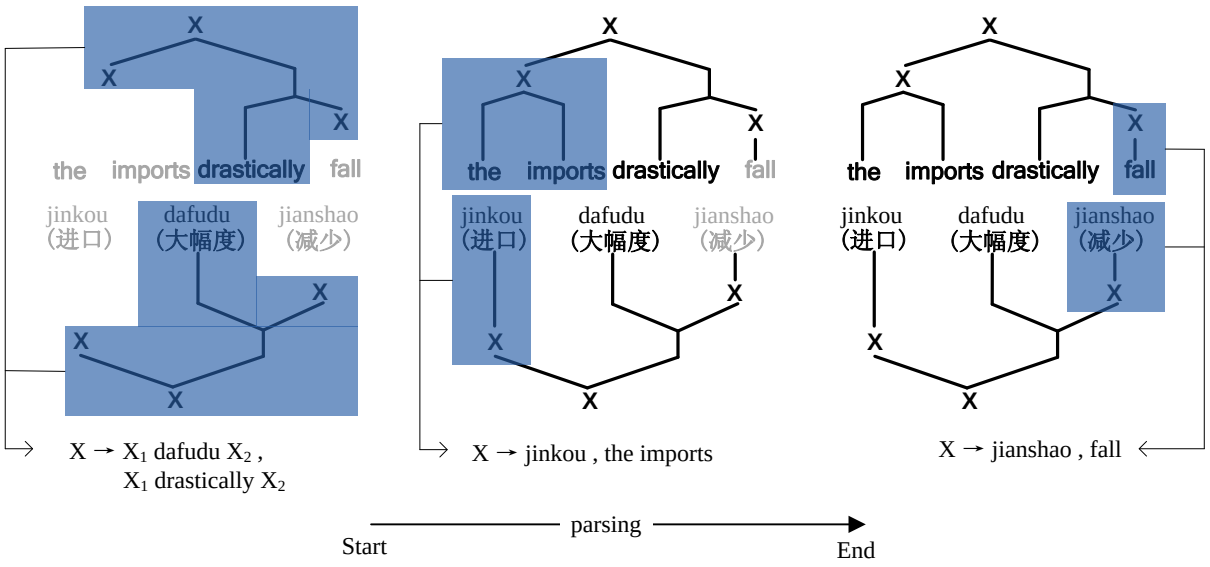


Figure 4.2. Sample derivation of the hierarchical phrase-based model

$$\text{NP} \longrightarrow \text{jinkou , the imports} \quad \text{(S4)}$$

$$\text{VB} \longrightarrow \text{jianshao , fall} \quad \text{(S5)}$$

$$\text{S} \longrightarrow \text{NP}_1 \text{ dafudu VB}_2 \text{ , NP}_1 \text{ drastically VB}_2 \quad \text{(S6)}$$

4.1.2.2 Introducing Real Syntax with Tree Structures

Obviously, the rules described above explain the mapping from source-language strings to (simplified) target-language syntax, and thus can be viewed as the instances of grammar rules used string-to-tree models. However, in the SCFG formulism, the target-language output of grammar rule is a string which cannot express the tree-fragment encoded in a general string-to-tree model. To obtain tree-formatted output in target-language, we need to introduce tree representation into the outputs of grammar rules. For example, rules S4-S6 can be rewritten by adding target-language tree structures.

$$\text{NP} \longrightarrow \text{jinkou , DT(the) NNS(imports)} \quad \text{(S7)}$$

$$\text{VB} \longrightarrow \text{jianshao , fall} \quad \text{(S8)}$$

$$\text{S} \longrightarrow \text{NP}_1 \text{ dafudu VB}_2 \text{ , NP}_1 \text{ VP(RB(drastically) VB}_2\text{)} \quad \text{(S9)}$$

The above rules are standard rules used in string-to-tree translation where an input string is mapped into a target-language pattern (or subtree). In general, they can be represented using the xRs transducers [Galley et al., 2006]. Here we use a similar way as that used in the xRs transducers to represent them, as follows

$$\text{jinkou} \Longrightarrow \text{NP(DT(the) NNS(imports))} \quad \text{(S10)}$$

$$\text{jianshao} \implies \text{VB(fall)} \tag{S11}$$

$$\text{NP}_1 \text{ dafudu VB}_2 \implies \text{S(NP}_1 \text{ VP(RB(drastically) VB}_2\text{))} \tag{S12}$$

where $\Longrightarrow$ separates the source and target-language sides of the rule. In some cases, xRs rule r (or SCFG rule) is also represented as a tuple $(s(r), t(r), \phi(r))$, where $s(r)$ is the source-language side of r (i.e., left part of the rule), $t(r)$ is the target-language side of r (i.e., right part of the rule), and $\phi(r)$ is the alignments of variables between two languages. For example, for rule S12, we have:

$$\begin{aligned} s(r) &= \text{NP :}x \text{ dafudu VB :}x \\ t(r) &= \text{S(NP :}x \text{ VP(RB(drastically) VB :}x\text{))} \\ \phi(r) &= \{1-1, 2-2\} \end{aligned}$$

where x marks the variable in the rule, and $\phi(r)$ is a set of one-to-one alignments that link up source non-terminal (indexing from 1) and target non-terminal (indexing from 1).

Note that the representation of xRs rules does not follow the framework of SCFG strictly. In other words, SCFG rules and xRs rules may result in different formalizations of translation process. For example, the derivations generated using rules S4-S6 (SCFG rules) and rules S10-S12 (xRs rules) are different, though the same target-language syntax is provided (See Figure 4.3). Fortunately, in practical systems, different rule representations do not always result in changes of translation accuracy. Actually, the systems based on these two grammar formalisms are of nearly the same performance in our experiments.

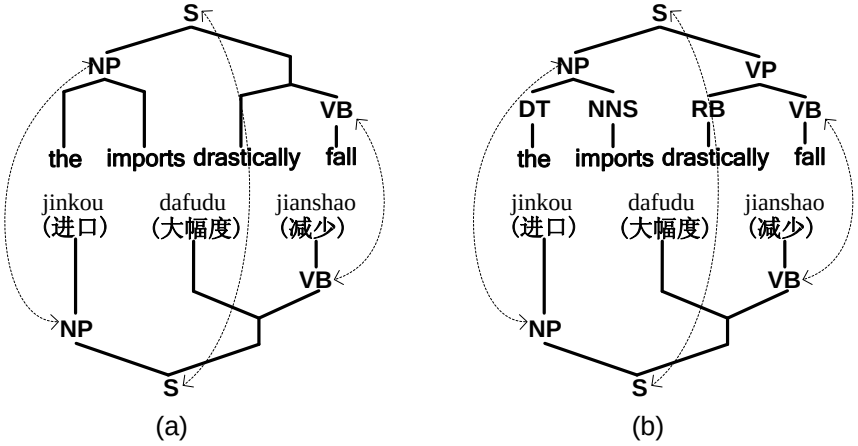


Figure 4.3. Comparison of derivations generated using SCFG rules (a) and xRs rules (b). The dotted lines link the non-terminals that are rewritten in parsing.

In addition to the string-to-tree model, the grammar rules of the tree-to-string model can also be represented by SCFG or xRs transducers. However, the xRs representation does not fit for the tree-to-tree model as the source-language side is a tree-fragment instead of a string. In this case, grammar rules in tree-to-tree translation are generally expressed by Synchronous Tree-Substitution Grammars (STSGs). In STSG, both the source and target-language sides are represented as tree-fragments. Such a way of representation is very useful in handling the transformation from a source tree to a target tree, as in tree-to-tree translation. To illustrate STSG more clearly, a few STSG rules are shown as follows. Further,

Figure 4.4 depicts a sample (tree-to-tree) derivation generated using these rules.

$$\text{NP(NN(jinkou))} \implies \text{NP(DT(the) NNS(imports))} \quad \text{(S13)}$$

$$\text{VV(jianshao)} \implies \text{VB(fall)} \quad \text{(S14)}$$

$$\text{S(NP}_1\ \text{VP(AD(dafudu) VV}_2\text{))} \implies \text{S(NP}_1\ \text{VP(RB(drastically) VB}_2\text{))} \quad \text{(S15)}$$

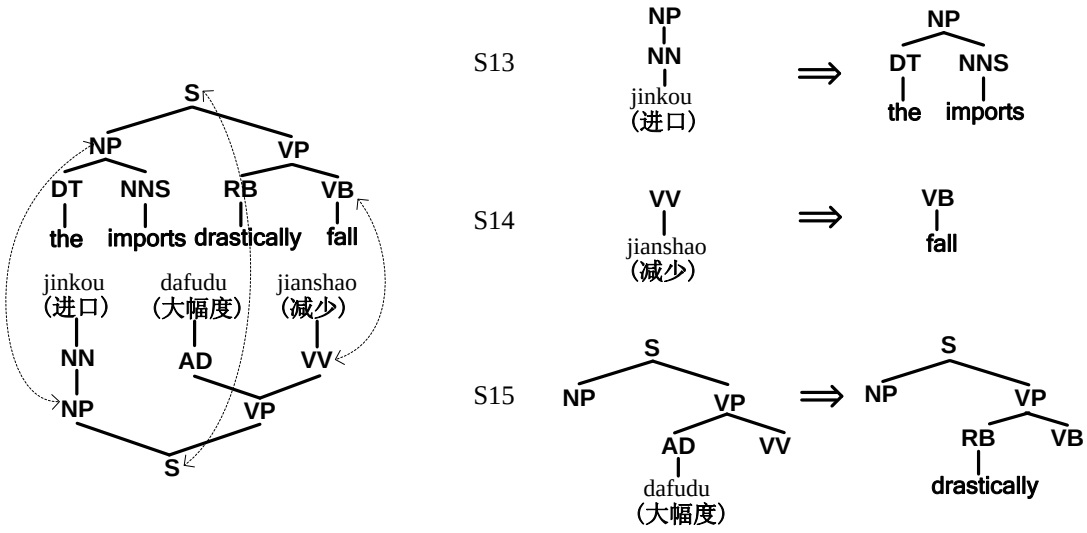


Figure 4.4. Sample derivation of tree-to-tree MT

It is worth noting that SCFG, xRs transducers and STSG are all standard instances of the general framework of synchronous grammar despite of differences in detailed formulism. Therefore, they share most properties of synchronous grammars and are weekly equivalent when applied to MT.

4.1.3 Grammar Induction

Like phrase-based MT, syntax-based MT requires a "table" of translation units which can be accessed in the decoding stage to form translation derivations. So the first issue in syntax-based MT is to learn such a table from bilingual corpus. Different approaches are adopted for the hierarchical phrase-based model and the syntax-based models.

4.1.3.1 Rule Extraction for Hierarchical Phrase-based Translation

We first present how synchronous grammar rules are learned according to the hierarchical phrase-based model. Here we choose the method proposed in [Chiang, 2005]. In [Chiang, 2005], it is assumed that there is no underlying linguistic interpretation and the non-terminal is labeled with X only.

Given a collection of word-aligned sentence pairs, it first extracts all phrase-pairs that are consistent with the word alignments, as in standard phrased-based models (See Section 3.1.3). The extracted phrase-pairs are the same as those used phrase-based MT. In the hierarchical phrase-based model, they are generally called *phrasal rules* or *traditional phrase translation rules*. Figure 4.5(a) shows an example of

extracting phrase rules from a word-aligned sentence pair. In hierarchical phrase-based MT, these rules are also written in the standard form of SCFG. Like this

$$\text{X} \longrightarrow \text{zai zhuozi shang , on the table} \tag{S16}$$

$$\text{X} \longrightarrow \text{zai zhuozi shang de , on the table} \tag{S17}$$

$$\text{X} \longrightarrow \text{zai zhuozi shang de pingguo , the apple on the table} \tag{S18}$$

...

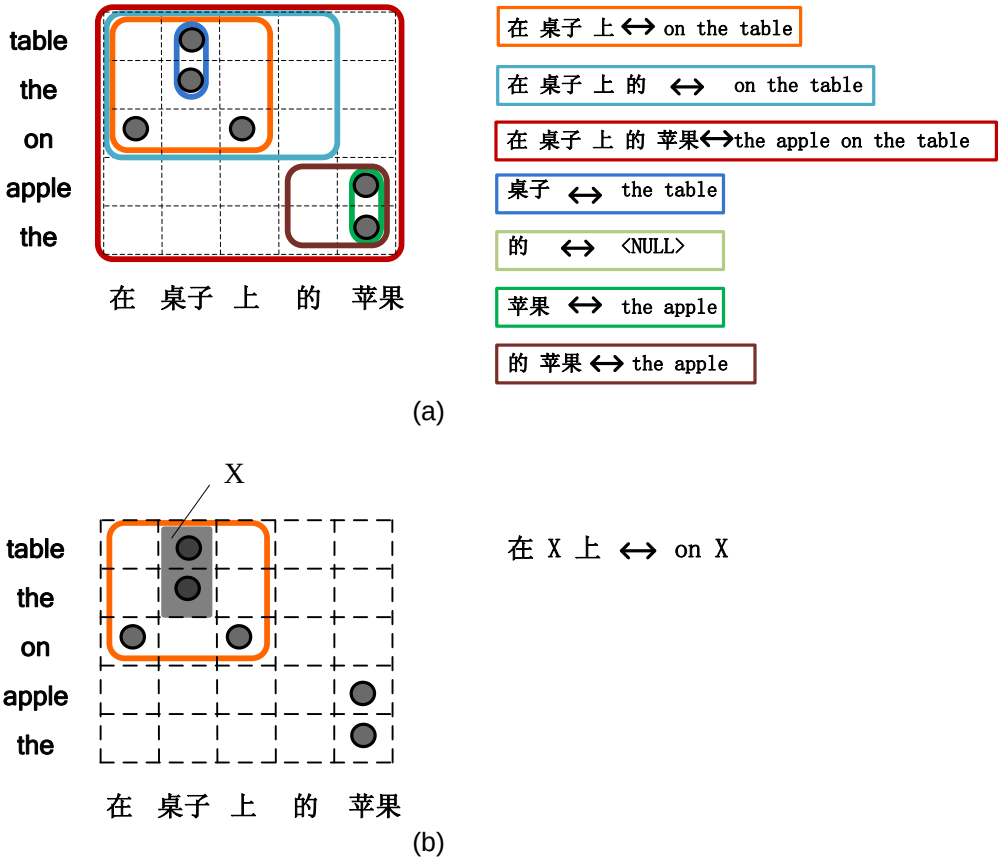


Figure 4.5. Example of extracting traditional phrase-pairs and hierarchical phrase rules

Then, we learn more complex rules that involve both terminal and nonterminal (variable) symbols on the right hand side of the rule. See Figure 4.5(a) for an example. Obviously, traditional phrase extraction is not able to handle the discontinues phrases in which some internal words (or intervening words) are generalized to be a "slot", such as "zai ... shang". In this case, we need to learn as generalizations of the traditional phrasal rules. To do this, we first replace the internal words *zhuozi* with the non-terminal symbol X on the source-language side, and then replace *the table* on the target-language side accordingly. As a result, we obtain a new rule that contains sub-blocks that can be replaced with symbol X. For more intuitive understanding, we list a few more rules that represent the hierarchical phrase structures, as

follows:

$$\text{X} \longrightarrow \text{zai X}_1\text{ shang , on X}_1 \quad \text{(S19)}$$

$$\text{X} \longrightarrow \text{zai zhuozi shang de X}_1\text{ , X}_1\text{ on the table} \quad \text{(S20)}$$

$$\text{X} \longrightarrow \text{zai X}_1\text{ shang de X}_2\text{ , X}_2\text{ on X}_1 \quad \text{(S21)}$$

...

Note that the number of possible rules is exponential to the number of words in the input sentences. In general, we need to introduce some constraints into rule extraction to avoid an unmanageable rule set. As suggested in [Chiang, 2005], ones may consider the following limits

- no consecutive non-terminals are allowed
- at most 2 non-terminals appear on each language side
- rules are extracted on spans having at most 10 words

Another note on rule induction. In [Chiang, 2005], a special rule, *glue rule*, is defined to directly compose the translations of adjunct spans, as an analogy to traditional phrase-based approaches. This rule has been proved to be very useful in improving hierarchical phrase-based systems, and thus is used under the default setting of NiuTrans.Hierarchy.

4.1.3.2 Syntactic Translation Rule Extraction

We have described a method that learns synchronous grammar rules without any truly syntactic annotation. In this section, we consider how to add syntax into translation rules, as what we expect in syntax-based MT.

As syntactic information is required in rule extraction, syntax trees of the training sentences should be repaired before extraction. Generally, the syntax trees are automatically generated using syntactic parsers[3]. Here we suppose that the target-language parse trees are available. Next, we will describe a method to learn translation rules for the string-to-tree model.

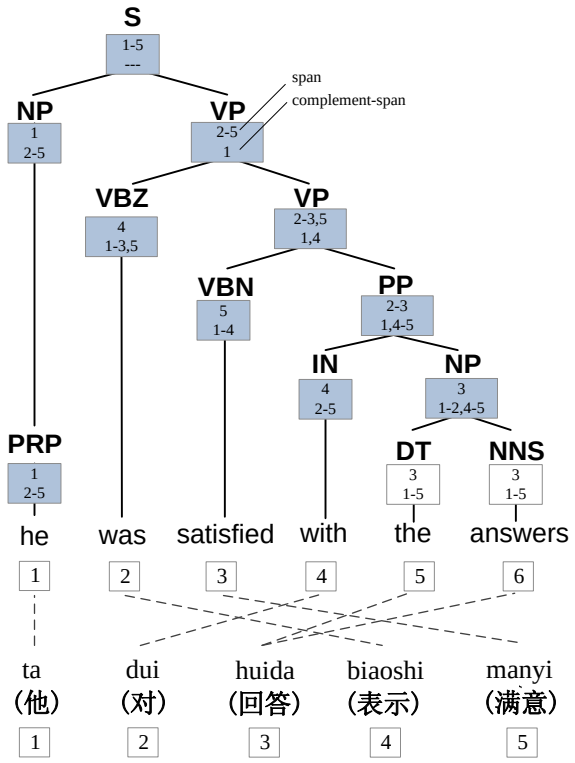
In the syntax-based engine of NiuTrans, the basic method of rule extraction is the so-called GHKM extraction [Galley et al., 2006]. The GHKM method is developed for learning syntactic translation rules from word-aligned sentence pairs whose target-language (or source-language) side has been already parsed. The idea is pretty simple: we first compute the set of the minimally-sized translation rules that can explain the mappings between source-language string and target-language tree while respecting the alignment and reordering between the two languages, and then learn larger rules by composing two or more minimal rules.

Recall that, in the previous section, all hierarchical phrase rules are required to be consistent with the word alignments. For example, any variable in a hierarchical phrase rule is generalized from a valid phrase-pair that does not violate any word alignments. In the GHKM extraction, all syntactic translation rules follow the same principle of alignment consistency. Beyond this, the rules are learned respecting the

[3]To date, several open-source parsers have been developed and achieved state-of-the-art performance on the Penn Treebank for several languages, such as Chinese and English.

target-language syntax tree. That is, the target-side of the resulting rules is a tree-fragment of the input parse tree. Before introducing the GHKM algorithm, let us consider a few concepts which will be used in the following description.

The input of GHKM extraction is a tuple of source string, target tree and alignments between source and target terminals. The tuple is generally represented as a graph (See Figure 4.6 for an example). On each node of the target tree, we compute the values of *span* and *complement span*. Given a node u in the target tree, $span(u)$ is defined as the set of words in the source string that are reachable from u. $complement{-}span(u)$ is defined as the union set of all spans of the nodes that are neither u's descendants nor ancestors. Further, u is defined to be an *admissible* node if and only if $complement{-}span(u) \cap span(u) = \varnothing$. In Figure 4.6, all nodes in shaded color are admissible nodes (each is labeled with the corresponding values of $span(u)$ and $complement{-}span(u)$). The set of admissible nodes is also called as *frontier set* and denoted as F. According to [Galley et al., 2006], the major reason for defining the frontier set is that, *for any frontier of the graph containing a given node $u \in F$, spans on that frontier define an ordering between u and each other frontier node u'*. For example, admissible node PP(4-6) does not overlap with (but precedes or follows) other nodes. However, node NNS(6-6) does not hold this property.



Translation Rules Extracted

r_1 : ta → **NP**(**PRP** (he))

r_2 : dui → **IN** (with)

r_3 : huida → **NP** (**DT**(the) **NNS**(answers))

r_4 : biaoshi → **VBZ** (was)

r_5 : manyi → **VBN** (satisfied)

r_6 : **IN**$_1$ **NP**$_2$ → **PP** (**IN**$_1$ **NP**$_2$)

r_7 : dui **NP**$_1$ → **PP** (**IN**(with) **NP**$_1$)

r_8 : **NP**$_1$ **PP**$_2$ biaoshi manyi →
S (**NP**$_1$ **VP**(**VBZ**(was) **VP**(**VBN**(satisfied) **PP**$_2$)))

⋮

Figure 4.6. Example of string-tree graph and rules extracted

As the frontier set defines an ordering of constituents, it is reasonable to extract rules by ordering constituents along sensible frontiers. To realize this idea, the GHKM extraction considers the rules whose target-language side matches only the admissible nodes defined in the frontier set. For example, r_6 in Figure 4.6 is a valid rule according to this definition since all the variables of the right-hand side correspond to admissible nodes.

Under such a definition, the rule extraction is very simple. First, we extract all minimal rules that cannot be decomposed into simpler rules. To do this, we visit each node u of the tree (in any order) and extract the minimal rule rooting at u by considering both the nearest descendants of u and the frontier set. Then, we can compose two or more minimal rules to form larger rules. For example, in Figure 4.6, r_{1-6} are minimal rules, while r_7 is a composed rule generated by combining r_2 and r_6.

Obviously, the above method is directly applicable to the tree-to-string model. Even when we switch to tree-to-tree translation, this method still works fine by extending the frontier set from one language side to both language sides. For tree-to-tree rule extraction, what we need is to visit each pair of nodes, instead of the nodes of the parse on one language side as in the original GHKM algorithm. On each node pair (u, v) we enumerate all minimal tree-fragments rooting at u and v according to the bilingual frontier set. The minimal rules are then extracted by aligning the source tree-fragments to the target tree-fragments, with the constraint that the extracted rules does not violate word alignments. The larger rules can be generated by composing minimal rules, which is essentially the same procedure of rule composing in the GHKM extraction.

4.1.4 Features Used in NiuTrans.Hierarchy/NiuTrans.Syntax

The hierarchical phrase-based and syntax-based engines adopts a number of features to model derivation's probability. Some of them are inspired by the phrase-based model, the others are designed for the hierarchical phrase-based and syntax-based systems only. The following is a list of the features used in NiuTrans.Hierarchy/NiuTrans.Syntax.

Basic Features (for both hierarchical phrase-based and syntax-based engines)

- **Phrase-based translation probability** $\Pr(\tau_t(r)|\tau_s(r))$. In this document $\tau(\alpha)$ denotes a function that returns the frontier sequence of the input tree-fragment α[4]. Here we use $\tau_s(r)$ and $\tau_t(r)$ to denote the frontier sequences of source and target-language sides. For example, for rule r_7 in Figure 4.6, the frontier sequences are

$$\begin{aligned} \tau_s(r) &= \text{dui NP} \\ \tau_t(r) &= \text{with NP} \end{aligned}$$

  $\Pr(\tau_t(r)|\tau_s(r))$ can be obtained by relative frequency estimation, as in Equation 3.9.

- **Inverted phrase-based translation probability** $\Pr(\tau_s(r)|\tau_t(r))$. The inverted version of $\Pr(\tau_t(r)|\tau_s(r))$.

- **Lexical weight** $\Pr_{lex}(\tau_t(r)|\tau_s(r))$. The same feature as that used in the phrase-base system (see Section 3.10).

- **Inverted lexical weight** $\Pr_{lex}(\tau_s(r)|\tau_t(r))$. The inverted version of $\Pr_{lex}(\tau_t(r)|\tau_s(r))$.

- **N-gram language model** $\Pr_{lm}(\mathbf{t})$. The standard n-gram language model.

[4] If α is already in string form, $\tau(\alpha) = \alpha$

- **Target word bonus (TWB)** $length(\mathbf{t})$. It is used to eliminate the bias of n-gram LM which prefers shorter translations.
- **Rule bonus (RB)**. This feature counts the number of rules used in a derivation. It allows the system to learn a preference for longer or shorter derivations.
- **Word deletion bonus (WDB)**. This feature counts the number of word deletions (or explicit null-translations) in a derivation. It allows the system to learn how often word deletion is performed.

Syntax-based Features (for syntax-based engine only)

- **Root Normalized Rule Probability** $\Pr(r|root(r))$. Here $root(r)$ denotes the root symbol of rule r. $\Pr(r|root(r))$ can be computed using relative frequency estimation:

$$\Pr(r|root(r)) = \frac{count(r)}{\sum_{root(r')=root(r)}(r')} \tag{4.1}$$

- **IsComposed** $IsComposed(r)$. A indicator feature function that has value 1 for composed rules, 0 otherwise.
- **IsLexicalized** $IsLex(r)$. A indicator feature function that has value 1 for lexicalized rules, 0 otherwise.
- **IsLowFrequency** $IsLowFreq(r)$. A indicator feature function that has value 1 for low-frequency rules (appear less than 3 times in the training corpus), 0 otherwise.

Then, given a derivation d and the corresponding source-string $\mathbf{s}$ and target-string $\mathbf{t}$, $Pr(\mathbf{t}, d|\mathbf{s})$ is computed as follows

$$\Pr(\mathbf{t}, d|\mathbf{s}) = \prod_{r \in d} score(r) \times \Pr_{lm}(\mathbf{t})^{\lambda_{lm}} \times \exp(\lambda_{TWB} \cdot length(\mathbf{t}))/Z(\mathbf{s}) \tag{4.2}$$

where $Z(\mathbf{s})$ is the normalization factor and can be ignored when searching for the best derivation with maximum probability $\Pr(\mathbf{t}, d|\mathbf{s})$. Following the framework of weighted synchronous grammar [Aho and Ullman, 1969], we use $\prod_{r \in d} score(r)$ to estimate the goodness of the derivation, and assign a score (or weight) $score(r)$ to each grammar rule r with a log-linear model

$$\begin{aligned} score(r) = {} & \Pr(\tau_t(r)|\tau_s(r))^{\lambda_1} \times \Pr(\tau_s(r)|\tau_t(r))^{\lambda_2} \times \Pr_{lex}(\tau_t(r)|\tau_s(r))^{\lambda_3} \times \Pr_{lex}(\tau_s(r)|\tau_t(r))^{\lambda_4} \times \\ & \Pr(r|root(r))^{\lambda_{root}} \times \\ & \exp(\lambda_{RB}) \times \exp(\lambda_{WDB} \cdot \delta(\bar{s} \rightarrow null)) \times \\ & \exp(\lambda_{IsComposed(r)}) \times \exp(\lambda_{IsLex(r)}) \times \exp(\lambda_{IsLowFreq(r)}) \end{aligned} \tag{4.3}$$

Like NiuTrans.Phrase, all the feature weights ($\{\lambda\}$) of NiuTrans.Hierarchy/NiuTrans.Syntax are optimized on a development data-set using minimum error rate training.

4.1.5 Decoding as Chart Parsing

4.1.5.1 Decoding with A Sample Grammar

In principle, decoding with a given SCFG/STSG can be cast as a parsing problem, which results in different decoding algorithms compared to phrase-based models. For example, we cannot apply the left-to-right decoding method to handle synchronous grammars since the gaps in grammar rules would produce discontinues target-language words.

On the other hand, the left-hand side of synchronous grammar rules always cover valid constituents, which motivates us to recursively build derivations (and corresponding sub-trees) by applying those grammar rules in a bottom-up fashion. In other words, when applying the constraint of (single) constituent to the input language, we can represent the input sentence as a tree structure where each constituent covers a continuous span. In NiuTrans, we choose *chart parsing* to realize this process. The key idea of chart parsing is to decode along (continuous) spans of the input sentence. We start with initializing the chart by lexicalized rules covering continuous word sequence. Larger derivations are then built by applying grammar rules to composing those derivations of the smaller chart entries. The decoding process completes when the algorithm covers the entire span. Figure 4.7 illustrates the chart parsing algorithm with an example derivation.

Given the input sentence and seven grammar rules, the algorithm begins with translating source words into target words. In this example, we can directly translate *ta*, *huida*, *biaoshi* and *manyi* using four purely lexicalized rules r_1 and r_{3-5} (or phrasal rules) where no variables are involved. When these rules are mapped onto the input words, we build (target) tree structure accordingly. For example, when *ta* is covered by rule r_1 : ta $\longrightarrow$ NP(PRP(he)), we build the corresponding (target-language) tree structure NP(PRP(he)). Similarly, we can build the target sub-trees NP(DT(the) NNS(answers)), VBZ(was) and VBN(satisfied) using rules r_{3-5}. Note that, in practical systems, we may obtain many grammar rules that match the same source span and produce a large number of competing derivations in the same chart cell during decoding. Here we simply ignore competing rules in this example. The issue will be discussed in the following parts of this section.

We then switch to larger spans after processing the spans covering only one word. Only rule r_2 can be applied to spans of length two. Since *huida* has been already translated into NP(DT(the) NNS(answers)), we can apply the following rule to span *dui huida*.

$$\text{dui NP}_1 \longrightarrow \text{PP(IN(with) NP}_1$$

where non-terminal NP_1 matches the chart entry that has already been processed (i.e., entry of span *huida*). When the rule applies, we need to check the label of the chart entry (i.e., NP) to make sure that the label of matched non-terminal is consistent with the chart entry label. Then we build a new chart entry which contains the translation of *dui huida* and the pointers to previous chart entries that are used to build it.

Next, we apply the rule

$$\text{PP}_1 \text{ VBZ}_2 \text{ VBN}_3 \longrightarrow \text{VP(VBZ}_2 \text{ VP(VBN}_3 \text{ PP}_1\text{)}$$

Grammar Rules

r_1 : ta → **NP**(**PRP** (he))

r_2 : dui **NP**$_1$ → **PP**(**IN** (with) **NP**$_1$)

r_3 : huida → **NP** (**DT**(the) **NNS**(answers))

r_4 : biaoshi → **VBZ** (was)

r_5 : manyi → **VBN** (satisfied)

r_6 : **PP**$_1$ **VBZ**$_2$ **VBN**$_3$ → **VP**(**VBZ**$_2$ **VP**(**VBN**$_3$ **PP**$_1$))

r_7 : **NP**$_1$ **VP**$_2$ → **NP**$_1$ **VP**$_2$

Chart

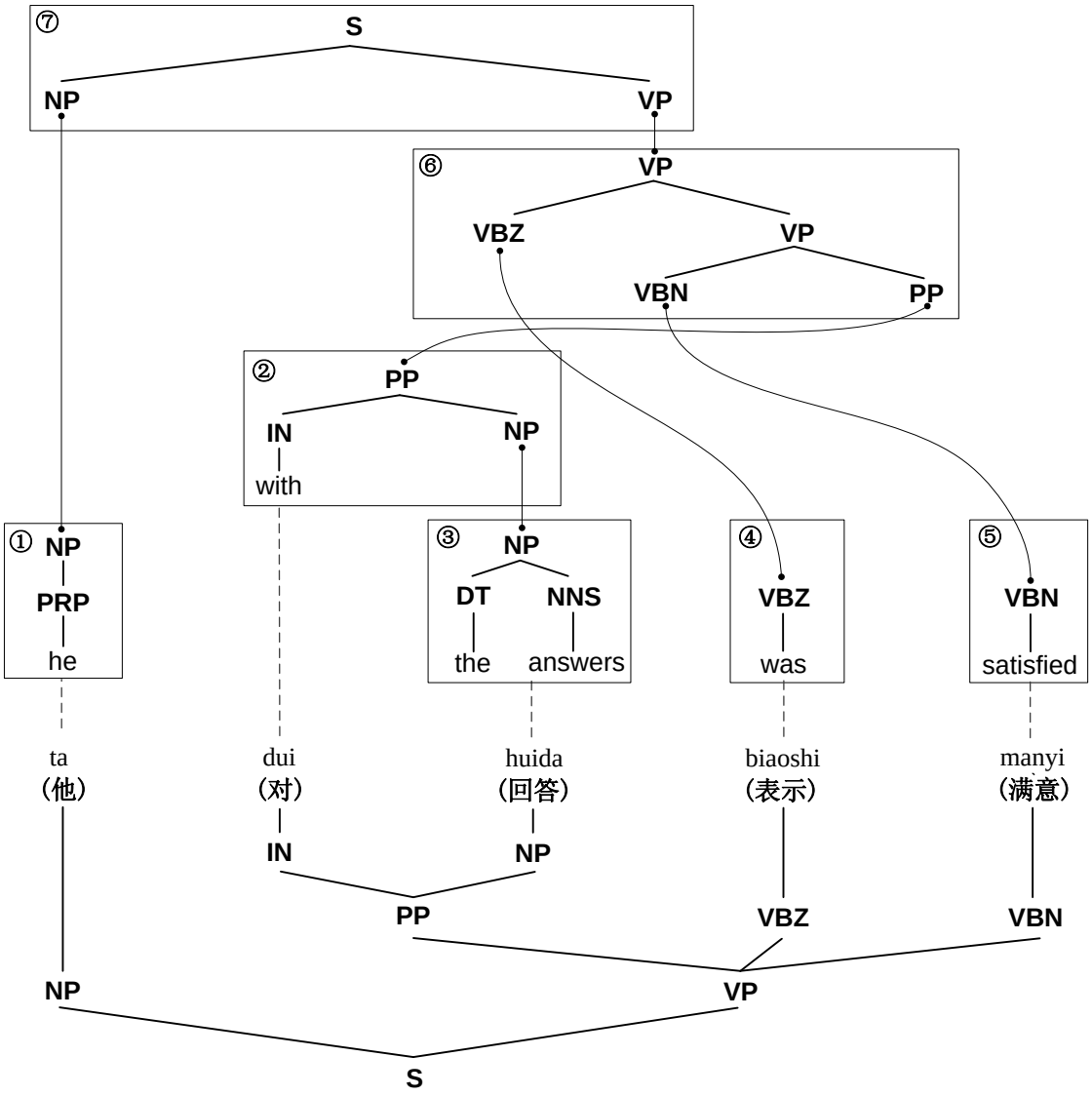


Figure 4.7. Sample derivation generated using the chart parsing algorithm.

It covers the span of four words *dui huida biaoshi manyi*. This rule is a non-lexicalized rule and does not have any terminals involved. It contains three variables PP, VBZ and VBN, which hold different positions in input and output languages. Thus the rule application causes the reordering of *was satisfied* and *the answer*.

At last, we apply the following rule in the same way.

$$\text{NP}_1\ \text{VP}_2 \longrightarrow \text{S}(\text{NP}_1\ \text{VP}_2)$$

This rule covers the entire span and creates a chart entry that completes the translation from the input string to the translation.

4.1.5.2 Algorithm

As described above, given a source sentence, the chart-decoder generates 1-best or k-best translations in a bottom-up manner. The basic data structure used in the decoder is a chart, where an array of cells is organized in topological order. Each cell maintains a list of items (chart entries). The decoding process starts with the minimal cells, and proceeds by repeatedly applying translation rules to obtain new items. Once a new item is created, the associated scores are computed (with an integrated n-gram language model). Then, the item is added into the list of the corresponding cell. This procedure stops when we reach the final state (i.e., the cell associates with the entire source span). The decoding algorithm is sketched out in Figure 4.8.

The chart decoding algorithm

Input: source string $\mathbf{s} = s_1 ... s_J$, and the synchronous grammar G
Output: (1-best) translation

```
1:  Function ChartDecoding(s, G)
2:    for j1 = 1 to J do                          ▷ beginning of span
3:      for j2 = j1 to J do                       ▷ ending of span
4:        foreach r in G do                       ▷ consider all the grammar rules
5:          foreach sequence s of words and chart entries in span[j1, j2] do
                                                  ▷ consider all the patterns
6:            if r is applicable to s do
7:              h =CreateHypo(r, s)               ▷ create a new item
8:              cell[j1, j2].Add(h)               ▷ add a new item into the candidate list
9:    return cell[1, J].1best()
```

Figure 4.8. The chart decoding algorithm

For a given sentence of length n, there are $n(n-1)/2$ chart cells. As (real-world) synchronous grammars may provide many translations for input words or patterns, there is generally an extremely large number of potential items that can be created even for a single chart cell. Therefore, we need to carefully organize the chart structure to make the decoding process tractable.

Generally, we need a priority queue to record the items generated in each span. The main advantage of using this structure is that we can directly perform *beam search* by keeping only the top-k items in the

priority queue. Also, this data structure is applicable to other advanced pruning methods, such as cube pruning.

When a new item is created, we need to record 1) the partial translation of the corresponding span; 2) the root label of the item (as well as the grammar rule used); 3) backward pointers to other items that used to construct it; and 4) the model score of the item. All this information is associated with the item and can be accessed in the later steps of decoding. Obviously, such a record encodes the path (or derivation) the decoder generates. By tracking the backward pointers, we can easily recover the derivation of grammar rules used in generating the translation.

When we judge whether a item can be used in a specific rule application, we only need to check the span and root label of the item. It is reasonable to organize the priority queues based on the span they cover. Alternatively, we can organize the priority queues based on both the coverage span and root label. In this way, only the items sharing the same label would compete with each other, and the system can benefit from the less competition of derivations and fewer search errors. As a "penalty", we need to maintain a very large number of priority queues and have to suffer from lower decoding speed. In NiuTrans, we implement the chart structure and priority queues using the first method due to its simplicity. See Figure 4.11 for an illustration of the organization of the chart structure, as well as how the items are built according to the algorithm described above.

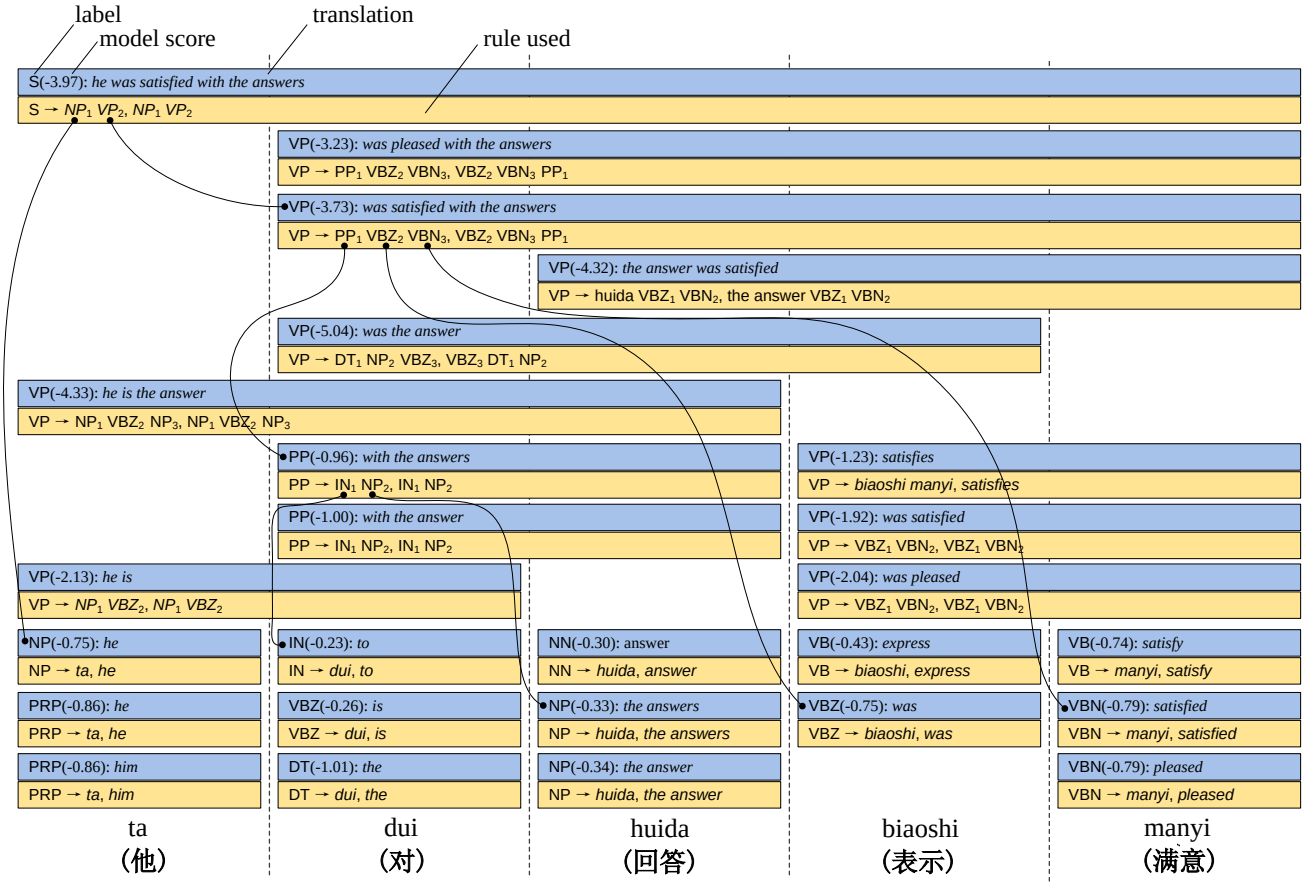


Figure 4.9. Some of the chart cells and items generated using chart parsing (for string-to-tree translation). The round-head lines link up the items that are used to construct the (1-best) derivation.

4.1.5.3 Practical Issues

To build an efficient decoder, several issues should be further considered in the implementation.

Pruning. Like phrase-based systems, syntax-based systems requires pruning techniques to obtain acceptable translation speed. Due to the more variance in underlying structures compared to phrase-based systems, syntax-based systems generally confront a more severe search problem. In NiuTrans, we consider both *beam pruning* and *cube pruning* to make decoding computational feasible. We implement beam pruning using the histogram pruning method. Its implementation is trivial: once all the items of the cell are proved, only the top-k best items according to model score are kept and the rest are discarded. Cube pruning is essentially an instance of heuristic search, which explores the most "promising" candidates based on the previous searching path. Here we do not present the details about cube pruning. Readers can refer to [Chiang, 2007] for a detailed description.

Binarization. As described previously, decoding with a given SCFG/STSG is essentially a (monolingual) parsing problem, whose complexity is in general exponential in the number of non-terminals on the right-hand side of grammar rules [Zhang et al., 2006]. To alleviate this problem, two solutions are available. The simplest of these is that we restrict ourselves to a simpler grammar. For example, in the Hiero system [Chiang, 2005], the source-language side of all SCFG rules is restricted to have no adjunct frontier non-terminals and at least one terminal on the source-language side. However, syntax-based systems achieve excellent performance when they use flat n-ary rules that have many non-terminals and models very complex translation phenomena [DeNero et al., 2010]. To parse with all available rules, a more desirable solution is *grammar transformation* or *grammar encoding* [Zhang et al., 2006; DeNero et al., 2010]. That is, we transform the SCFG/STSG into an equivalent binary form. Consequently, the decoding can be conducted on a binary-branching SCFG/STSG with a "CKY-like" algorithm. For example, the following is a grammar rule which is flat and have more than two non-terminals.

$$\begin{aligned} \text{S} \longrightarrow\ & \text{zhexie yundongyuan AD VV NP he NP,} \\ & \text{DT these players VB coming from NP and NP} \end{aligned}$$

It can be binarized into equivalent binary rules, as follows:

$$\begin{aligned} \text{S} &\longrightarrow \text{V}^1\ \text{NP},\ \ \text{V}^1\ \text{NP} \\ \text{V}^1 &\longrightarrow \text{V}^2\ \text{he},\ \ \text{V}^2\ \text{and} \\ \text{V}^2 &\longrightarrow \text{V}^3\ \text{NP},\ \ \text{V}^3\ \text{NP} \\ \text{V}^3 &\longrightarrow \text{V}^4\ \text{laizi},\ \ \text{V}^4\ \text{comingfrom} \\ \text{V}^4 &\longrightarrow \text{V}^5\ \text{VV},\ \ \text{V}^5\ \text{VB} \\ \text{V}^5 &\longrightarrow \text{zhexie yundongyuan AD}_1,\ \ \text{DT}_1\ \text{these players} \end{aligned}$$

Then decoding can proceed as usual, but with some virtual non-terminals (V^{1-5}). In this document we do not discuss the binarizaion issue further. Please refer to [Zhang et al., 2006] for more details.

Hypothesis Recombination. Another issue is that, for the same span, there are generally items that have the same translation and the same root label, but with different underlying structures (or decoding

paths). In a sense, this problem reflects some sort of *spurious ambiguity*. Obviously it makes no sense to record all these equivalent items. In NiuTrans, we eliminate those equivalent items by keeping only the best item (with highest model score). Under such a way, the system can generate more diverse translation candidates and thus choose "better" translations from a larger pool of unique translations.

4.1.6 Decoding as Tree-Parsing

While treating MT decoding as a parsing problem is a natural solution to syntax-based MT, there are alternative ways to decode input sentence when source-language parse trees are provided. For example, in the tree-to-string model, all source-side parse trees[5] are available in either rule extraction or decoding stage. In this case, it is reasonable to make better use of the input parse tree for decoding, rather than the input word sequence only. Decoding from input parse has an obvious advantages over the string-parsing counterpart: the input tree can help us prune the search space. As we only need to consider the derivations that match the (source-language) tree structure, many derivations are ruled out due to their "incompatible" (source-language) structures. As a result, the explored derivation space shrinks greatly and the decoder only searches over a very small space of translation candidates. On the other hand, this decoding method suffers from more search errors in spite of a great speed improvement. In general, decoding with the input parse tree degrades in translation accuracy, but the performance drop varies in different cases, for example, for Chinese-English news-domain translation, the use of the input parse tree can provide stable speed improvements but leads to slight decreases of BLEU score. However, for translation tasks of other language pairs, such a method still suffers from a relatively lower BELU score.

The tree parsing algorithm

Input: the source parse tree S, and the synchronous grammar G
Output: (1-best) translation

```
1:  Function TreeParsing(S, G)
2:    foreach node v ∈ S in top-down order do        ▷ traverse the tree
3:      foreach r in G do                            ▷ consider all the grammar rules
4:        if MatchRule(r, v, S) = true do            ▷ map the rule onto the tree node
5:          S[v].Add(r)
6:    foreach node v ∈ S in bottom-up order do       ▷ traverse the tree again
7:      foreach r in S[v] do                         ▷ loop for each matched rule
8:        h =CreateHypo(r, v, S)                     ▷ create an item
9:        cell[v].Add(h)                             ▷ add the new item into the candidate list
10:   return cell[root].1best()
11: Function MatchRule(r, v, S)
12:   if root(r) = v and s(r) is a fragment of tree S do return true
13:   else return false
```

Figure 4.10. The tree parsing algorithm

Generally the approach described above is called *tree-parsing* [Eisner, 2003]. In tree-parsing, translation rules are first mapped onto the nodes of input parse tree. This results in a translation tree/forest (or a

[5]Parse tree are generally generated using automatic parsers

hypergraph) where each edge represents a rule application. Then decoding can proceed on the hypergraph as usual. That is, we visit in bottom-up order each node in the parse tree, and calculate the model score for each edge rooting at the node. The final output is the 1-best/k-best translations maintained by the root node of the parse tree. See Figure 4.10 for the pseudo code of the tree-parsing algorithm. Also, we show an illustration of the algorithm for tree-to-tree translation in Figure 4.11. Note that tree-parsing differs from parsing only in the rule matching stage, and the core algorithm of decoding is actually more of the same. This means that, in tree-parsing, we can re-use the pruning and hypothesis recombination components of the parsing-based decoder.

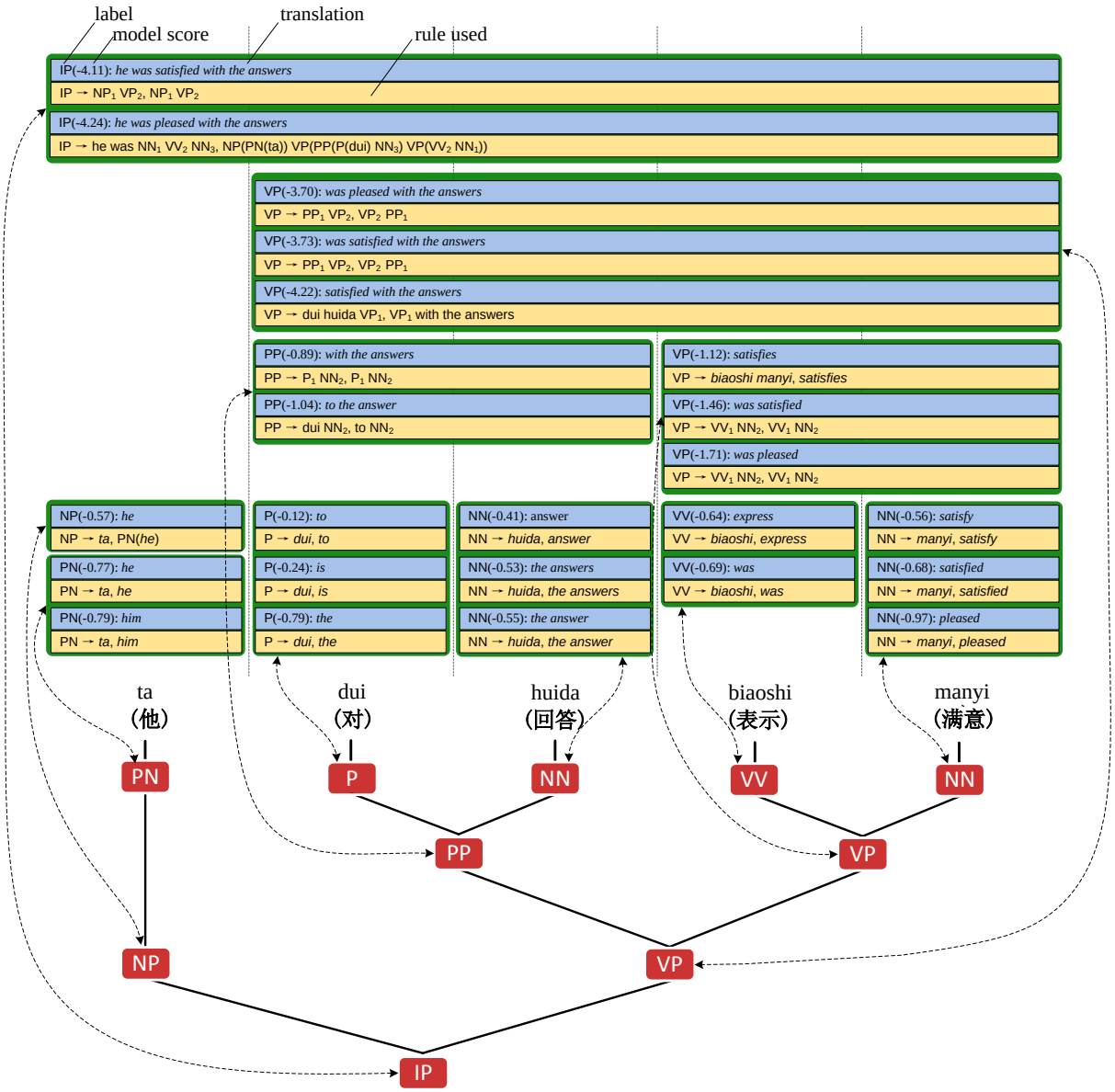


Figure 4.11. Some of the chart cells and items generated using tree parsing (for tree-to-string translation). The dashed lines link up the items and corresponding tree node of the input parse tree.

Another note on decoding. For tree-based models, *forest-based decoding* [Mi et al., 2008] is a natural

extension of tree-parsing-based decoding. In principle, forest is a type of data structure that can encode exponential number of trees efficiently. This structure has been proved to be helpful in reducing the effects caused by parser errors. Since our internal representation is already in a hypergraph structure, it is easy to extend the decoder to handle the input parse forest, with little modification of the code.

4.2 Step 1 - Rule Extraction and Parameter Estimation

4.2.1 NiuTrans.Hierarchy

Next, we introduce the detailed instructions to set-up the *NiuTrans.Hierarchy* engine. We start with rule extraction and parameter estimation which are two early-stage components of the training pipeline. In *NiuTrans*, they are implemented in a single program, namely *NiuTrans.PhraseExtractor* (in /bin/). Basically, *NiuTrans.PhraseExtractor* have four functions which corresponds to the four steps in rule extraction and parameter estimation.

- **Step 1**: Extract hierarchical phrase-pairs from word-aligned sentence-pairs.
- **Step 2**: Extract lexical translations from word-aligned sentence-pairs (for calculating lexical weights. See Section 3.1.5).
- **Step 3**: Obtain the associated scores for each hierarchical phrase-pair.
- **Step 4**: Filter the hierarchical-rule table.

4.2.1.1 Rule Extraction

As described above, the first step is learning hierarchical phrase translations from word-aligned bilingual corpus. To extract various hierarchical phrase-pairs (for both source-to-target and target-to-source directions), the following command is used in NiuTrans:

Command

```
$ cd NiuTrans/bin/
$ mkdir ../work/hierarchical.rule/ -p
$ ./NiuTrans.PhraseExtractor --EXTH \
      -src   ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tgt   ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -aln   ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -out   ../work/hierarchical.rule/hierarchical.phrase.pairs
```

where the following options MUST be specified:

`--EXTH`, which indicates that the program (NiuTrans.PhraseExtractor) works for extracting hierarchical phrase-pairs.

`-src`, which specifies the source-language side of the training data (one sentence per line).

`-tgt`, which specifies the target-language side of the training data (one sentence per line).

`-aln`, which specifies the word alignments between the source and target sentences.

`-out`, which specifies file of extracted hierarchical phrase pairs.

Output: two files "hierarchical.phrase.pairs" and "hierarchical.phrase.pairs.inv" are generated in "/NiuTrans/work/hierarchical.rule/".

Output (/NiuTrans/work/hierarchical.rule/)

```
- hierarchical.phrase.pairs          ▷ "source → target" hierarchical phrases
- hierarchical.phrase.pairs.inv      ▷ "target → source" hierarchical phrases
```

4.2.1.2 Obtaining Lexical Translation

As two lexical weights are involved in the NiuTrans system (See $Pr_{lex}(\tau_t(r)|\tau_s(r))$ and $Pr_{lex}(\tau_s(r)|\tau_t(r))$ in Section 4.1.4), lexical translations are required before parameter estimation. The following instructions show how to obtain lexical translation file (in both source-to-target and target-to-source directions) in the NiuTrans system:

Command

```
$ cd NiuTrans/bin/
$ mkdir ../work/lex/ -p
$ ./NiuTrans.PhraseExtractor --LEX \
      -src   ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tgt   ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -aln   ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -out   ../work/lex/lex
```

where

`--LEX`, which indicates that the program (NiuTrans.PhraseExtractor) works for extracting lexical translations.

`-src`, which specifies the source sentences of bilingual training corpus.

`-tgt`, which specifies the target sentences of bilingual training corpus.

`-aln`, which specifies word alignments between the source and target sentences.

`-out`, which specifies the prefix of output files (i.e., lexical translation files)

Also, there are some optional parameters, as follows:

`-temp`, which specifies the directory for sorting temporary files generated during the processing.

`-stem`, which specifies whether stemming is used. e.g., if `-stem` is specified, all the words are stemmed.

Output: two files "lex.s2d.sorted" and "lex.d2s.sorted" are generated in "/NiuTrans/work/lex/".

Output (/NiuTrans/work/lex/)

```
- lex.s2d.sorted        ▷ "source → target" lexical translation file
- lex.d2s.sorted        ▷ "target → source" lexical translation file
```

4.2.1.3 Generating Hierarchical-Rule Table

The next step is the generation of hierarchical-rule table which will then be used in the following decoding steps. Basically the hierarchical-rule table is a collections of hierarchical phrase-pairs with associated scores (or features). In NiuTrans, all the hierarchical phrase-pairs are sorted in alphabetical order, which makes the system can efficiently loads/organizes the hierarchical-rule table in a internal data structure. Each entry of the table is made up several fields. To illustrate their meaning, See Figure 4.12 for a fragment of hierarchical-rule table.

Hierarchical-Rule Table

```
...
#X yao #X ||| the #2 of #1 ||| X ||| -1.20397 -4.12004 0 -2.59355 1 0
#X liangan #X ||| #1 cross - strait #2 ||| X ||| 0 -4.58482 0 -0.723998 1 0
#X de #X . ||| #2 of #1 . ||| X ||| -1.60944 -2.10718 0 -1.58197 1 0
yige zhongguo ||| one china ||| X ||| 0 -1.72565 0 -1.63656 1 0
yixie rencai ||| some qualified personnel ||| X ||| -1.09861 -4.42797 -0.693147 -2.18392 1 0
bubian he dangjiazuozhu ||| unchanged and to be masters ||| X ||| 0 -7.64573 0 -4.34477 1 0
zhonggong zhongyang ||| the cpc central committee ||| X ||| -1.09861 -5.67531 0 -2.84051 1 0
...
```

Figure 4.12. Example of hierarchical-rule table

In this example, each line is separated into four fields using " ||| ". The meaning of them are:

- The **first** field is the source side of the hierarchical phrase-pair.
- The **second** field is the target side of the hierarchical phrase-pair.
- The **third** field is the left-hand side of Synchronous CFG.
- The **forth** field is the set of features associated with the entry. The first four features are $\Pr(\tau_t(r)|\tau_s(r))$, $\Pr_{lex}(\tau_t(r)|\tau_s(r))$, $\Pr(\tau_s(r)|\tau_t(r))$, and $\Pr_{lex}(\tau_s(r)|\tau_t(r))$ (See Section 4.1.4). The 5th feature is the phrase bonus exp(1). The 6th is an "undefined" feature which is reserved for feature engineering and can be defined by users.

Then, the following instructions can be adopted to generate the hierarchical-rule table from extracted hierarchical phrases and lexical translation tables:

Command

```
$ cd NiuTrans/bin/
$ ./NiuTrans.PhraseExtractor --SCORE \
      -tab      ../work/hierarchical.rule/hierarchical.phrase.pairs \
      -tabinv   ../work/hierarchical.rule/hierarchical.phrase.pairs.inv \
      -ls2d     ../work/lex/lex.s2d.sorted \
      -ld2s     ../work/lex/lex.d2s.sorted \
      -out      ../work/hierarchical.rule/hierarchical.rule.step1
```

where

`--SCORE` indicates that the program (NiuTrans.PhraseExtractor) runs in the "scoring" mode. It scores each hierarchical phrase-pairs, removes the replicated entries, and sort the table.

`-tab` specifies the file of extracted hierarchical phrases in "source → target" direction.

`-tabinv` specifies the file of extracted hierarchical phrases in "target → source" direction.

`-ls2d` specifies the lexical translation table in "source → target" direction.

`-ld2s` specifies the lexical translation table in "target → source" direction.

`-out` specifies the resulting hierarchical-rule table.

The optional parameters are:

`-cutoffInit` specifies the threshold for cutting off low-frequency initial phrase-pairs. e.g., "`-cutoffInit`=1" means that the program would ignore the initial phrase-pairs that appear only once, while "`-cutoffInit`=0" means that no initial phrases are discarded.

`-cutoffHiero` specifies the threshold for cutting off low-frequency hierarchical phrase-pairs.

`-printFreq` specifies whether the frequency information (the 5th field) is outputted.

`-printAlign` specifies whether the alignment information (the 6th field) is outputted.

`-temp` specifies the directory for sorting temporary files generated in the above procedure.

Output: in this step one file are generated under "/NiuTrans/work/hierarchical.rule/"

Output (/NiuTrans/work/hierarchical.rule/)

```
- hierarcical.rule.step1        ▷ hierarchical rule table
```

4.2.1.4 Hierarchical-Rule Table Filtering

In NiuTrans, the maximum number of translation options (according to $\Pr(\tau_t(r)|\tau_s(r))$) can be set by users (See following instructions).

Command

```
$ cd NiuTrans/bin/
$ ./NiuTrans.PhraseExtractor --FILTN \
      -in             ../work/hierarchical.rule/hierarchical.rule.step1 \
      -out            ../work/hierarchical.rule/hierarchical.rule \
      -strict         30 \
      -tableFormat    hierarchy
```

where

`--FILTN` indicates that we run the program (NiuTrans.PhraseExtractor) to filtering the hierarchical-rule table.

`-in` specifies the input file (i.e., the hierarchical-rule table)

`-out` specifies the output file (i.e., the filtered hierarchical-rule table)

`-strict` specifies the maximum number of translation options for each source-side of hierarchical-rule (30 by default).

`-tableFormat` specifies the format of input table (i.e., the value of this parameter is "phrase", "hierarchy" or "syntax").

Output: the filtered table ("hierarchical.rule") is placed in "NiuTrans/work/hierarchical.rule/". It will be used as a sample hierarchical-rule table in the following illustration in this section.

Output (/NiuTrans/work/hierarchical.rule/)

```
- hierarchical.rule  ▷ (filtered) hierarchical-rule table for the following steps
```

4.2.2 NiuTrans.Syntax

Here we describe how to set-up the *NiuTrans.Syntax* engine. We start with rule extraction and parameter estimation which are two early-stage components of the training pipeline. In *NiuTrans*, they are implemented in two programs, namely *NiuTrans.SyntaxRuleEx* and *NiuTrans.PhraseExtractor* (in /bin/). Basically, *NiuTrans.SyntaxRuleEx* have one function which corresponds to the first step in rule extraction, and *NiuTrans.PhraseExtractor* have three functions which corresponds to the following steps in parameter estimation.

- **Step 1**: Extract syntax-rules from word-aligned sentence-pairs and source and traget parse trees.
- **Step 2**: Extract lexical translations from word-aligned sentence-pairs (for calculating lexical weights. See Section 3.1.5).
- **Step 3**: Obtain the associated scores for each syntax-rule.
- **Step 4**: Filter the scored syntax-rule table.

4.2.2.1 Rule Extraction

As described above, the first step is learning syntax-rule translations from word-aligned bilingual corpus and source and target parse trees. To extract syntax-rule (for string-to-tree, tree-to-string and tree-to-tree model), the following command is used in NiuTrans (ones can select one of them according their models):

Command (string-to-tree)

```
$ cd NiuTrans/bin/
$ mkdir ../work/syntax.string2tree/ -p
$ ./NiuTrans.SyntaxRuleEx \
      -model      s2t \
      -method     GHKM \
      -src        ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tar        ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -align      ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -tarparse   ../sample-data/sample-submission-version/TM-training-set/english.tree.txt \
      -output     ../work/syntax.string2tree/syntax.string2tree.rule
```

Command (tree-to-string)

```
$ cd NiuTrans/bin/
$ mkdir ../work/syntax.tree2string/ -p
$ ./NiuTrans.SyntaxRuleEx \
      -model      t2s \
      -method     GHKM \
      -src        ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tar        ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -align      ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -srcparse   ../sample-data/sample-submission-version/TM-training-set/chinese.tree.txt \
      -output     ../work/syntax.tree2string/syntax.tree2string.rule
```

Command (tree-to-tree)

```
$ cd NiuTrans/bin/
$ mkdir ../work/syntax.tree2tree/ -p
$ ./NiuTrans.SyntaxRuleEx \
      -model      t2t \
      -method     GHKM \
      -src        ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tar        ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -align      ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -srcparse   ../sample-data/sample-submission-version/TM-training-set/chinese.tree.txt \
      -tarparse   ../sample-data/sample-submission-version/TM-training-set/english.tree.txt \
      -output     ../work/syntax.tree2tree/syntax.tree2tree.rule
```

where

`-model`, specify SMT translation model, the model decides what type of rules can be extracted, its value can be "s2t", "t2s" or "t2t", default "t2s".

`-method`, specify rule extraction method, its value can be "GHKM" or "SPMT", default "GHKM".

`-src`, specify path to the source sentence file.

`-tar`, specify path to the target sentence file.

`-align`, specify path to the word alignment file.

`-srcparse`, specify path to the source sentence parse tree file, the parse tree format is link Berkeley Parser's output.

`-tarparse`, specify path to the target sentence parse tree file, the parse tree format is like Berkeley Parser's output.

`-output`, specify path to the output file, default "stdout".

Also, there are some optional parameters, as follows:

`-inverse`, extract inversed language-pair rules.

`-compose`, specify the maximum compose times of atom rules, the atom rules are either GHKM minimal admissible rule or lexical rules of SPMT Model 1.

`-varnum`, specify the maximum number of variables in a rule.

`-wordnum`, specify the maximum number of words in a rule.

`-uain`, specify the maximum number of unaligned words in a rule.

`-uaout`, specify the maximum number of unaligned words outside a rule.

`-depth`, specify the maximum depth of tree in a rule.

`-oformat`, specify the format of generated rule, its value can be "oft" or "nft", default "nft".

Output: Each executed command generates one file in corresponding directory.

Output (rule for string-to-tree model in /NiuTrans/work/syntax.string2tree/)

```
- syntax.string2tree.rule            ▷ string-to-tree syntax rule
```

Output (rule for tree-to-string model in /NiuTrans/work/syntax.tree2string/)

```
- syntax.tree2string.rule            ▷ tree-to-string syntax rule
```

Output (rule for tree-to-tree model in /NiuTrans/work/syntax.tree2tree/)

```
- syntax.tree2tree.rule              ▷ tree-to-tree syntax rule
```

4.2.2.2 Obtaining Lexical Translation

As two lexical weights are involved in the *NiuTrans* system (See $\Pr_{lex}(\tau_t(r)|\tau_s(r))$ and $\Pr_{lex}(\tau_s(r)|\tau_t(r))$ in Section 4.1.4), lexical translations are required before parameter estimation. The following instructions show how to obtain lexical translation file (in both source-to-target and target-to-source directions) in the *NiuTrans* system:

Command

```
$ cd NiuTrans/bin/
$ mkdir ../work/lex/ -p
$ ./NiuTrans.PhraseExtractor --LEX \
      -src   ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tgt   ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -aln   ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -out   ../work/lex/lex
```

where

`--LEX`, which indicates that the program (NiuTrans.PhraseExtractor) works for extracting lexical translations.

`-src`, which specifies the source sentences of bilingual training corpus.

`-tgt`, which specifies the target sentences of bilingual training corpus.

`-aln`, which specifies word alignments between the source and target sentences.

`-out`, which specifies the prefix of output files (i.e., lexical translation files)

Also, there are some optional parameters, as follows:

`-temp`, which specifies the directory for sorting temporary files generated during the processing.

`-stem`, which specifies whether stemming is used. e.g., if `-stem` is specified, all the words are stemmed.

Output: two files "lex.s2d.sorted" and "lex.d2s.sorted" are generated in "/NiuTrans/work/lex/".

Output (/NiuTrans/work/lex/)

```
- lex.s2d.sorted        ▷ "source → target" lexical translation file
- lex.d2s.sorted        ▷ "target → source" lexical translation file
```

4.2.2.3 Generating Syntax-Rule Table

The next step is the generation of syntax-rule table which will then be used in the following decoding steps. Basically the rule table is a collections of syntax-rule with associated scores (or features). In *NiuTrans*, all the syntax-rules are sorted in alphabetical order, which makes the system can efficiently loads/organizes the rule table in a internal data structures. Each entry of the rule table is made up several fields. To illustrate their meaning, Figure 4.13, 4.14 and 4.15 shows three sample tables for different model.

Syntax-Rule Table (string-to-tree)

```
...
#ADJP de ziyuan ||| #1 resources ||| NP ||| -0.693147 -0.297569 0 -1.97069 1 -8.10682 -8.10682 1 1 1 0 ||| 0-0 2-1
#ADJP er #ADJP ||| #1 and #2 ||| ADJP ||| 0 -1.56134 -0.693147 -5.19099 1 -4.82831 -4.82831 1 1 1 0 ||| 0-0 1-1 2-2
#ADVP jiejue ||| resolved #1 ||| VP ||| -0.693147 -2.3186 0 -0.313139 1 -7.48773 -6.79459 1 1 1 0 ||| 0-1 1-0
...
```

Figure 4.13. Example of syntax-rule table for string-to-tree model

Syntax-Rule Table (tree-to-string)

```
...
#ADJP zhishi ||| #1 knowledge ||| NP ||| 0 -0.263861 0 -0.538997 1 -8.80822 -8.80822 1 1 1 0 ||| 0-0 1-1
#ADJP zuguo ||| #1 motherland ||| NP ||| -1.09861 -0.835236 0 -0.127955 1 -8.80822 -7.70961 1 1 1 0 ||| 0-0 1-1
#DNP renwu ||| #1 mission ||| NP ||| -0.693147 -2.71328 0 -1.39747 1 -8.80822 -8.11507 1 1 1 0 ||| 0-0 1-1
...
```

Figure 4.14. Example of syntax-rule table for tree-to-string model

Syntax-Rule Table (tree-to-tree)

```
...
#CD=QP yishang ||| more than #1 ||| NP=LCP ||| 0 -3.4185 0 -3.34901 1 -2.56495 -2.56495 1 1 1 0 ||| 0-2 1-0 1-1
#DT=DP qu ||| #1 zone ||| NP=NP ||| -0.405465 -2.15211 0 -1.20734 1 -6.04619 -6.04619 1 1 1 0 ||| 0-0 1-1
#DT=DT  ||| #1 years ||| NP=DP ||| 0 -0.84161 0 -0.612879 1 -1.60944 -1.60944 1 1 1 0 ||| 0-0 1-1
...
```

Figure 4.15. Example of syntax-rule table for tree-to-tree model

In this example, each line is separated into five fields using " ||| ". The meaning of them are:

- The **first** field is the source side of syntax-rule.
- The **second** field is the target side of syntax-rule.
- The **third** field is the root label of syntax-rule.
- The **forth** field is the set of features associated with the entry. The first four features are $Pr(\tau_t(r)|\tau_s(r))$, $Pr_{lex}(\tau_t(r)|\tau_s(r))$, $Pr(\tau_s(r)|\tau_t(r))$, and $Pr_{lex}(\tau_s(r)|\tau_t(r))$ (See Section 4.1.4). The 5th feature is the phrase bonus exp(1). The 6th and 7th features are Root Normalized Rule Probability $Pr(r|root(r))$ and $Pr(\tau_s(r)|root(r))$. The 8th feature is a indicator feature function that has value 1 for lexicalized rules, 0 otherwise. The 9th feature is a indicator feature function that has value 1 for composed rules, 0 otherwise. The 10th feature is a indicator feature function that has value 1 for low-frequency rules (appear less than 3 times in the training corpus as default), 0 otherwise. The 11th is undefined.
- The **fifth** field is the word alignment between the source and target side of syntax-rule.

Then, the following instructions can be adopted to generate the scored syntax-rule table from extracted syntax-rules and lexical translation tables:

Command (string-to-tree)

```
$ cd NiuTrans/bin/
$ ./NiuTrans.PhraseExtractor --SCORESYN \
      -model      s2t \
      -ls2d       ../work/lex/lex.s2d.sorted \
      -ld2s       ../work/lex/lex.d2s.sorted \
      -rule       ../work/syntax.string2tree/syntax.string2tree.rule \
      -out        ../work/syntax.string2tree/syntax.string2tree.rule.scored
```

Command (tree-to-string)

```
$ cd NiuTrans/bin/
$ ./NiuTrans.PhraseExtractor --SCORESYN \
      -model      t2s \
      -ls2d       ../work/lex/lex.s2d.sorted \
      -ld2s       ../work/lex/lex.d2s.sorted \
      -rule       ../work/syntax.tree2string/syntax.tree2string.rule \
      -out        ../work/syntax.tree2string/syntax.tree2string.rule.scored
```

Command (tree-to-tree)

```
$ cd NiuTrans/bin/
$ ./NiuTrans.PhraseExtractor --SCORESYN \
      -model      t2t \
      -ls2d       ../work/lex/lex.s2d.sorted \
      -ld2s       ../work/lex/lex.d2s.sorted \
      -rule       ../work/syntax.tree2tree/syntax.tree2tree.rule \
      -out        ../work/syntax.tree2tree/syntax.tree2tree.rule.scored
```

where

`--SCORESYN` indicates that the program (NiuTrans.PhraseExtractor) runs in the "syntax-rule scoring" mode. It scores each syntax-rule, removes the replicated entries, and sort the table.

`-model` specifies SMT translation model, the model decides what type of rules can be scored, its value can be "s2t", "t2s" or "t2t", default "t2s".

`-ls2d` specifies the lexical translation table in "source → target" direction.

`-ld2s` specifies the lexical translation table in "target → source" direction.

`-rule` specifies the extracted syntax-rule.

`-out` specifies the resulting hierarchical-rule table.

The optional parameters are:

`-cutoff` specifies the threshold for cutting off low-frequency syntax-rule. e.g., "`-cutoff` = 1" means that the program would ignore the syntax-rules that appear only once, while "`-cutoff` = 0" means that no syntax-rules are discarded.

`-lowerfreq` specifies the threshold for low-frequency, if the value set to 3, the syntax-rules which is appear less than 3 times are seen as low-frequency.

Output: in this step each scoring command generates one file in corresponding directory.

Output (rule table for string-to-tree model in /NiuTrans/work/syntax.string2tree/)

```
- syntax.string2tree.rule.scored        ▷ string-to-tree syntax rule table
```

Output (rule table for tree-to-string model in /NiuTrans/work/syntax.tree2string/)

```
- syntax.tree2string.rule.scored        ▷ tree-to-string syntax rule table
```

Output (rule table for tree-to-tree model in /NiuTrans/work/syntax.tree2tree/)

```
- syntax.tree2tree.rule.scored          ▷ tree-to-tree syntax rule table
```

4.2.2.4 Syntax-Rule Table Filtering

In *NiuTrans*, the maximum number of translation options (according to $Pr(\tau_t(r)|\tau_s(r))$) can be set by users (See following instructions). The filtering with test (or dev) sentences are not supported in the current version of the *NiuTrans* system.

Command (string-to-tree)

```
$ cd NiuTrans/bin/
$ ./NiuTrans.PhraseExtractor --FILTN \
      -in             ../work/syntax.string2tree/syntax.string2tree.rule.scored \
      -out            ../work/syntax.string2tree/syntax.string2tree.rule.scored.filter \
      -strict         30 \
      -tableFormat    syntax
$ cd ../scripts/
$ perl NiuTrans-change-syntaxrule-to-exp-format.pl \
      <     ../work/syntax.string2tree/syntax.string2tree.rule.scored.filter \
      >     ../work/syntax.string2tree/syntax.string2tree.rule.scored.filter.format
```

Command (tree-to-string)

```
$ cd NiuTrans/bin/
$ ./NiuTrans.PhraseExtractor --FILTN \
      -in             ../work/syntax.tree2string/syntax.tree2string.rule.scored \
      -out            ../work/syntax.tree2string/syntax.tree2string.rule.scored.filter \
      -strict         30 \
      -tableFormat    syntax
$ cd ../scripts/
$ perl NiuTrans-change-syntaxrule-to-exp-format.pl \
      <     ../work/syntax.tree2string/syntax.tree2string.rule.scored.filter \
      >     ../work/syntax.tree2string/syntax.tree2string.rule.scored.filter.format
```

Command (tree-to-tree)

```
$ cd NiuTrans/bin/
$ ./NiuTrans.PhraseExtractor --FILTN \
      -in            ../work/syntax.tree2tree/syntax.tree2tree.rule.scored \
      -out           ../work/syntax.tree2tree/syntax.tree2tree.rule.scored.filter \
      -strict        30 \
      -tableFormat   syntax
$ cd ../scripts/
$ perl NiuTrans-change-syntaxrule-to-exp-format.pl \
      <    ../work/syntax.tree2tree/syntax.tree2tree.rule.scored.filter \
      >    ../work/syntax.tree2tree/syntax.tree2tree.rule.scored.filter.format
```

where

`--FILTN` indicates that we run the program (NiuTrans.PhraseExtractor) to filtering the syntax-rule table.

`-in` specifies the input file (i.e., the syntax-rule table)

`-out` specifies the output file (i.e., the filtered syntax-rule table)

`-strict` specifies the maximum number of translation options for each source-side of syntax-rule (30 by default).

`-tableFormat` specifies the format of input table (i.e., the value of this parameter is "phrase", "hierarchy" or "syntax").

Output: each filter command generates one filtered rule table in the directory corresponding to the different models. The filtered file will be used as a sample syntax-rule table for different models in the following illustration in this section.

Output (filtered table for s2t model in /NiuTrans/work/syntax.string2tree/)

```
- syntax.string2tree.rule.scored.filter.format  ▷ (filtered) syntax-rule table
```

Output (filtered table for t2s model in /NiuTrans/work/syntax.tree2string/)

```
- syntax.tree2string.rule.scored.filter.format  ▷ (filtered) syntax-rule table
```

Output (filtered table for t2t model in /NiuTrans/work/syntax.tree2tree/)

```
- syntax.tree2tree.rule.scored.filter.format    ▷ (filtered) syntax-rule table
```

4.3 Step 2 - N-gram Language Modeling

The *NiuTrans* package offers a n-gram language modeling tool (*NiuTrans.LMTrainer*). This tool is placed in "NiuTrans/bin/". To train the n-gram language model, you can repeat the instructions shown in Section

2.2.

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/lm/ -p
$ perl NiuTrans-training-ngram-LM.pl \
      -corpus      ../sample-data/sample-submission-version/LM-training-set/e.lm.txt \
      -ngram       3 \
      -vocab       ../work/lm/lm.vocab \
      -lmbin       ../work/lm/lm.trie.data
```

where

`-corpus` specifies the training data file (i.e., a collection of target-language sentences. one sentence per line).

`-ngram` specifies the order of n-gram language model.

`-vocab` specifies the resulting vocabulary file.

`-lmbin` specifies the resulting model file.

Output: In the above example, two files (vocabulary and model file) are generated under "/NiuTrans/-work/lm/". They will be used the following steps of decoding.

Output (/NiuTrans/work/lm/)

```
- lm.vocab        ▷ vocabulary file
- lm.trie.data    ▷ model file of n-gram language model
```

4.4 Step 3 - Configuring the Decoder

4.4.1 NiuTrans.Hierarchy

4.4.1.1 Config File

Decoder is one of the most complicated components in modern SMT systems. Generally, many techniques (or tricks) are employed to successfully translate source sentences into target sentences. The *NiuTrans* system provides an easy way to set-up the decoder using a config file. Hence users can choose different settings by modifying this file and setup their decoders for different tasks. *NiuTrans*' config file follows the "key-value" definition. The following is a sample file which offers most necessary settings of the *NiuTrans.Hierarchy* system[6].

The meanings of these parameters are:

[6]Please see "/config/NiuTrans.hierarchy.config" for a more complete version of the config file

Decoder Config File (NiuTrans.Hierarchy)

```
param="Ngram-LanguageModel-File"    value="../sample-data/lm.trie.data"
param="Target-Vocab-File"           value="../sample-data/lm.vocab"
param="SCFG-Rule-Set"               value="../sample-data/hierarchical.rule"
param="nround"                      value="15"
param="ngram"                       value="3"
param="usepuncpruning"              value="1"
param="usecubepruning"              value="1"
param="nthread"                     value="4"
param="nbest"                       value="30"
param="outputnull"                  value="0"
param="beamsize"                    value="30"
param="nref"                        value="1"
param="fastdecoding"                value="1"
param="usenulltrans"                value="0"
param="snulltrans"                  value="1"
param="weights"                     value="2.000 1.000 1.000 0.200 1.000 0.200 \
                                           0.000 0.500 0.000 0.500 0.000 0.000 \
                                           0.000 0.000 0.000 0.000 0.000"
param="ranges"                      value="-3:7 -3:3 0:3 0:0.4 0:3 0:0.4 \
                                           -3:3 -3:3 -3:0 -3:3 -3:3 0:0 \
                                           0:0 0:0 0:0 0:0 0:0"
param="fixedfs"                     value="0 0 0 0 0 0 0 0 0 \
                                           0 0 0 0 0 0 0 0 0"
```

Figure 4.16. Decoder Config File (NiuTrans.Hierarchy)

- `Ngram-LanguageModel-File` specifies the n-gram language model file.
- `Target-Vocab-File` specifies the target-language vocabulary.
- `SCFG-Rule-Set` specifies the hierarchical-rule table.
- `nround` specifies how many rounds MERT performs. In each round of MERT run, the system produces the k-best translations and optimizes the feature weights.
- `ngram` specifies the order of n-gram language model used in decoding.
- `usepuncpruning` specifies whether the Punctuation Pruning is used (1: use punctuation pruning; 0: do not use it). If `usepuncpruning` is fired, the system would first divide the input sentence into smaller fragments according to punctuations (such as common). Then it decodes each fragment individually and glue their translations to generate the translation for the entire sentence.
- `usecubepruning` specifies whether the Cube Pruning is used (1: use cube pruning; 0: do not use it). For more details about cube pruning, please refer to [Huang and Chiang, 2005].

- `nthread` specifies the number of threads used in decoding source sentences. More threads means a higher speed. But, as most multi-thread programs, the speed improvement is very modest when a large number threads are involved. It is suggested to set `nthread` to $4 \sim 8$ on normal PC servers.

- `nbest` specifies the size of n-best list generated by the decoder. The direct use of n-best output is MERT which optimizes feature weights by promoting the "best-BLEU" candidate from n-best outputs of MT systems. Generally a large n-best list could result more stable convergence of MERT. However, a too large n-best does not really help.

- `outputnull` specifies whether OOV words and deleted words (null-translations) are outputted in final translations. When `outputnull` is fired, all those OOV or deleted words will be marked as "<something>". E.g., translation "I had a $< XX >$ day today!" indicates that XX is an OOV word or null-translation word that is deleted during decoding.

- `beamsize` specifies the size (or width) of beam used in beam search. A large beam could reduce the number of search errors, but in turn slows down the system.

- `nref` specifies how many reference translations are provided for MERT.

- `fastdecoding` speed-up the system.

- `usenulltrans` specifies whether explicit word deletion is allowed in decoding. If `usenulltrans = 1`, the decoder would delete some source words. Note that this feature is also called "devil feature" since it hurts the performance in some cases. e.g., in most applications, users do not expect to delete content words. However, the word-deletion feature does not consider such a factor. So users should be careful when using this feature.

- `snulltrans` allows sequence of null-translations.

- `weights` specifies the feature weights. In MERT, `weights` means the initial weights.

- `ranges` specifies the range (min and max values) for each individual feature during weight tuning. e.g., in the above example, the range of the first feature is "-3:7" which means that the compounding feature can only choose values over $[-3, 7]$.

- `fixedfs` specifies whether a feature weight is fixed (or not tuned) during MERT. "1" means the corresponding feature weight is fixed and not adjusted in weight tuning.

In *NiuTrans.Hierarchy* the features are ordered as follows (See fields `weights`, `ranges` and `fixedfs`)

id	feature	initial-value	min-value	max-value
1	n-gram language model	2.000	-3.000	7.000
2	target word bonus	1.000	-3.000	3.000
3	f $\rightarrow$ e translation probability	1.000	0.000	3.000
4	lexical weight	0.200	0.000	0.400
5	e $\rightarrow$ f translation probability	1.000	0.000	3.000
6	inverted lexical weight	0.200	0.000	0.400
7	rule bonus	0.000	-3.000	3.000
8	user-defined feature (for future extension)	0.500	-3.000	3.000
9	number of word-deletion operations	0.000	-3.000	0.000
10	number of phrasal rules	0.500	-3.000	3.000
11	number of glue rules	0.000	-3.000	3.000
12	undefined	0.000	0.000	0.000
13	undefined	0.000	0.000	0.000
14	undefined	0.000	0.000	0.000
15	undefined	0.000	0.000	0.000
16	undefined	0.000	0.000	0.000
17	undefined	0.000	0.000	0.000

4.4.1.2 Generating the Config File

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/config/ -p
$ perl NiuTrans-hierarchy-generate-mert-config.pl \
      -rule    ../work/hierarchical.rule/hierarchical.rule \
      -lmdir   ../work/lm/ \
      -nref    1 \
      -ngram   3 \
      -out     ../work/config/NiuTrans.hierarchy.user.config
```

where

`-rule` specifies the hierarchy rule table.

`-lmdir` specifies the directory that holds the n-gram language model and the target-side vocabulary.

`-nref` specifies how many reference translations per source-sentence are provided.

`-ngram` specifies the order of n-gram language model.

`-out` specifies the output (i.e. a config file).

Output: The output is file "NiuTrans.hierarchy.user.config" in "NiuTrans/work/config/". Users can modify "NiuTrans.hierarchy.user.config" as needed.

Output (NiuTrans/work/config/)

```
- NiuTrans.hierarchy.user.config        ▷ configuration file for MERT and decoding
```

4.4.2 NiuTrans.Syntax

4.4.2.1 Config File

Decoder is one of the most complicated components in modern SMT systems. Generally, many techniques (or tricks) are employed to successfully translate source sentences into target sentences. The *NiuTrans* system provides an easy way to set-up the decoder using a config file. Hence users can choose different settings by modifying this file and setup their decoders for different tasks. *NiuTrans*' config file follows the "key-value" definition. The following is a sample file which offers most necessary settings of the *NiuTrans.Syntax* system[7].

The meanings of these parameters are:

- `Ngram-LanguageModel-File` specifies the n-gram language model file.
- `Target-Vocab-File` specifies the target-language vocabulary.
- `SCFG-Rule-Set` specifies the syntax-rule table.
- `nround` specifies how many rounds MERT performs. In each round of MERT run, the system produces the k-best translations and optimizes the feature weights.
- `ngram` specifies the order of n-gram language model used in decoding.
- `usepuncpruning` specifies whether the Punctuation Pruning is used (1: use punctuation pruning; 0: do not use it). If `usepuncpruning` is fired, the system would first divide the input sentence into smaller fragments according to punctuations (such as common). Then it decodes each fragment individually and glue their translations to generate the translation for the entire sentence.
- `usecubepruning` specifies whether the Cube Pruning is used (1: use cube pruning; 0: do not use it). For more details about cube pruning, please refer to [Huang and Chiang, 2005].
- `nthread` specifies the number of threads used in decoding source sentences. More threads means a higher speed. But, as most multi-thread programs, the speed improvement is very modest when a large number threads are involved. It is suggested to set `nthread` to $4 \sim 8$ on normal PC servers.
- `nbest` specifies the size of n-best list generated by the decoder. The direct use of n-best output is MERT which optimizes feature weights by promoting the "best-BLEU" candidate from n-best outputs of MT systems. Generally a large n-best list could result more stable convergence of MERT. However, a too large n-best does not really help.

[7] Please see "/config/NiuTrans.syntax.s2t.config", "/config/NiuTrans.syntax.t2s.config" or "/config/NiuTrans.syntax.t2t.config" for a more complete version of the config file

Decoder Config File (NiuTrans.Syntax)

```
param="Ngram-LanguageModel-File"    value="../sample-data/lm.trie.data"
param="Target-Vocab-File"           value="../sample-data/lm.vocab"
param="SCFG-Rule-Set"               value="../sample-data/syntax.rule"
param="nround"                      value="15"
param="ngram"                       value="3"
param="usepuncpruning"              value="1"
param="usecubepruning"              value="1"
param="nthread"                     value="4"
param="nbest"                       value="30"
param="outputnull"                  value="0"
param="beamsize"                    value="30"
param="nref"                        value="1"
param="fastdecoding"                value="1"
param="beamscale"                   value="3"
param="usenulltrans"                value="0"
param="snulltrans"                  value="1"
param="incompletehyporate"          value="0.5"
param="weights"                     value="3.000 1.000 1.000 0.300 1.000 0.300 \
                                           0.000 1.000 -1.000 0.000 0.000 0.100 \
                                           0.100 1.000 0.000 -1.000 0.000"
param="ranges"                      value="-3:7 -3:3 0.5:3 0:0.4 0.5:3 0:0.4 \
                                           -3:3 -3:3 -3:0 0:0 -3:3 0.1:3 \
                                           0:0.2 -3:3 -3:3 -3:3 0:0"
param="fixedfs"                     value="0 0 0 0 0 0 0 0 0 \
                                           0 0 0 0 0 0 0 0 0"
```

Figure 4.17. Decoder Config File (NiuTrans.Hierarchy)

- `outputnull` specifies whether OOV words and deleted words (null-translations) are outputted in final translations. When `outputnull` is fired, all those OOV or deleted words will be marked as "<something>". E.g., translation "I had a $< XX >$ day today!" indicates that XX is an OOV word or null-translation word that is deleted during decoding.

- `beamsize` specifies the size (or width) of beam used in beam search. A large beam could reduce the number of search errors, but in turn slows down the system.

- `nref` specifies how many reference translations are provided for MERT.

- `fastdecoding` speed-up the system.

- `beamscale` scale beam width.

- `usenulltrans` specifies whether explicit word deletion is allowed in decoding. If `usenulltrans = 1`, the decoder would delete some source words. Note that this feature is also called "devil feature" since it hurts the performance in some cases. e.g., in most applications, users do not expect to delete

content words. However, this feature does not consider such a factor. So please be careful using this feature.

- `snulltrans` allows sequence of null-translations.
- `incompletehyporate` control the rate of incomplete states in beam search.
- `weights` specifies the feature weights. In MERT, `weights` means the initial weights.
- `ranges` specifies the range (min and max values) for each individual feature during weight tuning. e.g., in the above example, the range of the first feature is "-3:7" which means that the compounding feature can only choose values over $[-3, 7]$.
- `fixedfs` specifies whether a feature weight is fixed (or not tuned) during MERT. "1" means the corresponding feature weight is fixed and not adjusted in weight tuning.

The features used in *NiuTrans.Syntax* are listed as follows (See fields `weights`, `ranges` and `fixedfs`)

id	feature	initial-value	min-value	max-value
1	n-gram language model	3.000	-3.000	7.000
2	target word bonus	1.000	-3.000	3.000
3	f $\rightarrow$ e phrase-based probability	1.000	0.500	3.000
4	lexical weight	0.300	0.000	0.400
5	e $\rightarrow$ f phrase-based probability	1.000	0.500	3.000
6	inverted lexical weight	0.300	0.000	0.400
7	rule bonus	0.000	-3.000	3.000
8	user-defined feature (for future extension)	1.000	-3.000	3.000
9	number of word-deletion operations	-1.000	-3.000	0.000
10	number of phrasal rules	0.000	-3.000	3.000
11	number of glue rules	0.000	-3.000	3.000
12	root-normalized probability ($\Pr(r\|root(r))$)	0.100	0.100	3.000
13	source-side rule probability ($\Pr(shs(r)\|root(r))$)	0.100	0.000	0.200
14	number of lexicalized rules	1.000	-3.000	3.000
15	number of composed rules	0.000	-3.000	3.000
16	number of low-frequency rules	-1.000	-3.000	3.000
17	undefined	0.000	0.000	0.000

4.4.2.2 Generating the Config File

Command (string-to-tree)

```
$ cd NiuTrans/scripts/
$ mkdir ../work/config/ -p
$ perl NiuTrans-syntax-generate-mert-config.pl \
      -model          s2t \
      -syntaxrule     ../work/syntax.string2tree/syntax.string2tree.rule.scored.filter.format \
      -lmdir          ../work/lm/ \
      -nref           1 \
      -ngram          3 \
      -out            ../work/config/NiuTrans.syntax.s2t.user.config
```

Command (tree-to-string)

```
$ cd NiuTrans/scripts/
$ mkdir ../work/config/ -p
$ perl NiuTrans-syntax-generate-mert-config.pl \
      -model          t2s \
      -syntaxrule     ../work/syntax.tree2string/syntax.tree2string.rule.scored.filter.format \
      -lmdir          ../work/lm/ \
      -nref           1 \
      -ngram          3 \
      -out            ../work/config/NiuTrans.syntax.t2s.user.config
```

Command (tree-to-tree)

```
$ cd NiuTrans/scripts/
$ mkdir ../work/config/ -p
$ perl NiuTrans-syntax-generate-mert-config.pl \
      -model          t2t \
      -syntaxrule     ../work/syntax.tree2tree/syntax.tree2tree.rule.scored.filter.format \
      -lmdir          ../work/lm/ \
      -nref           1 \
      -ngram          3 \
      -out            ../work/config/NiuTrans.syntax.t2t.user.config
```

where

`-model` specifies what type of rules can be used to mert, its value can be ”s2t”, ”t2s” or ”t2t”.

`-syntaxrule` specifies the syntax-rule table.

`-lmdir` specifies the directory that holds the n-gram language model and the target-side vocabulary.

`-nref` specifies how many reference translations per source-sentence are provided.

`-ngram` specifies the order of n-gram language model.

`-out` specifies the output (i.e. a config file).

Output: The output is a file in "NiuTrans/work/config/". Users can modify this generated config file as needed.

Output (string-to-tree config file in NiuTrans/work/config/)

```
- NiuTrans.syntax.s2t.user.config        ▷ configuration file for MERT and decoding
```

Output (tree-to-string config file in NiuTrans/work/config/)

```
- NiuTrans.syntax.t2s.user.config        ▷ configuration file for MERT and decoding
```

Output (tree-to-tree config file in NiuTrans/work/config/)

```
- NiuTrans.syntax.t2t.user.config        ▷ configuration file for MERT and decoding
```

4.5 Step 4 - Weight Tuning

4.5.1 NiuTrans.Hierarchy

As the config is used to control the decoding and weight tuning processes, running MERT is very trivial in *NiuTrans*. Suppose that the config file is prepared, you can execute the following script to carry out the MER training.

Command

```
$ cd NiuTrans/scripts/
$ perl NiuTrans-hierarchy-mert-model.pl \
      -config    ../work/config/NiuTrans.hierarchy.user.config \
      -dev       ../sample-data/sample-submission-version/Dev-set/Niu.dev.txt \
      -nref      1 \
      -round     3 \
      -log       ../work/mert-model.log
```

where

`-config` specifies the configuration file generated in the previous steps.

`-dev` specifies the development dataset (or tuning set) for weight tuning.

`-nref` specifies how many reference translations per source-sentence are provided.

`-round` specifies how many rounds the MERT performs (by default, 1 round = 15 MERT iterations).

`-log` specifies the log file generated by MERT.

After MER training, the optimized feature weights are automatically recorded in "NiuTrans/work/-config/NiuTrans.hierarchy.user.config" (last line). Then, the config can be used to decode new sentences.

4.5.2 NiuTrans.Syntax

As the config is used to control the decoding and weight tuning processes, running MERT is very trivial in *NiuTrans*. Suppose that the config file is prepared, you can execute the following script to carry out the MER training.

Command (string-to-tree)

```
$ cd NiuTrans/scripts/
$ perl NiuTrans-syntax-mert-model.pl \
      -model     s2t \
      -config    ../work/config/NiuTrans.syntax.s2t.user.config \
      -dev       ../sample-data/sample-submission-version/Dev-set/Niu.dev.tree.txt \
      -nref      1 \
      -round     3 \
      -log       ../work/syntax-s2t-mert-model.log
```

Command (tree-to-string)

```
$ cd NiuTrans/scripts/
$ perl NiuTrans-syntax-mert-model.pl \
      -model     t2s \
      -config    ../work/config/NiuTrans.syntax.t2s.user.config \
      -dev       ../sample-data/sample-submission-version/Dev-set/Niu.dev.tree.txt \
      -nref      1 \
      -round     3 \
      -log       ../work/syntax-t2s-mert-model.log
```

Command (tree-to-tree)

```
$ cd NiuTrans/scripts/
$ perl NiuTrans-syntax-mert-model.pl \
      -model     t2t \
      -config    ../work/config/NiuTrans.syntax.t2t.user.config \
      -dev       ../sample-data/sample-submission-version/Dev-set/Niu.dev.tree.txt \
      -nref      1 \
      -round     3 \
      -log       ../work/syntax-t2t-mert-model.log
```

where

`-model` specifies what type of rules can be used to mert, its value can be "s2t", "t2s" or "t2t".

`-config` specifies the configuration file generated in the previous steps.

`-dev` specifies the development dataset (or tuning set) for weight tuning.

`-nref` specifies how many reference translations per source-sentence are provided.

`-round` specifies how many rounds the MERT performs (by default, 1 round = 15 MERT iterations).

`-log` specifies the log file generated by MERT.

After MER training, the optimized feature weights are automatically recorded in the "**-config**" file (last line). Then, the config can be used to decode new sentences.

4.6 Step 5 - Decoding

4.6.1 NiuTrans.Hierarchy

Last, users can decode new sentences with the trained model and optimized feature features[8]. The following instructions can be used:

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/hierarchy.trans.result/ -p
$ perl NiuTrans-hierarchy-decoder-model.pl \
      -config    ../work/config/NiuTrans.hierarchy.user.config \
      -test      ../sample-data/sample-submission-version/Test-set/Niu.test.txt \
      -output    ../work/hierarchy.trans.result/Niu.test.translated.en.txt
```

where

`-config` specifies the configuration file.

[8] you can still modify "NiuTrans.hierarchy.user.config" before testing

`-test` specifies the test dataset (one sentence per line).

`-output` specifies the translation result file (the result is dumped to "stdout" if this option is not specified).

Output: the (1-best) translation file "Niu.test.translated.en.txt" in "/NiuTrans/work/hierarchy.trans.result".

Output (NiuTrans/work/hierarchy.trans.result)

```
- Niu.test.translated.en.txt            ▷ 1-best translation of the test sentences
```

4.6.2 NiuTrans.Syntax

Last, users can decode new sentences with the trained model and optimized feature features[9]. The following instructions can be used:

Command (string-to-tree)

```
$ cd NiuTrans/scripts/
$ mkdir ../work/syntax.trans.result/ -p
$ perl NiuTrans-syntax-decoder-model.pl \
      -model     s2t
      -config    ../work/config/NiuTrans.syntax.s2t.user.config \
      -test      ../sample-data/sample-submission-version/Test-set/Niu.test.tree.txt \
      -output    ../work/syntax.trans.result/Niu.test.syntax.s2t.translated.en.txt
```

Command (tree-to-string)

```
$ cd NiuTrans/scripts/
$ mkdir ../work/syntax.trans.result/ -p
$ perl NiuTrans-syntax-decoder-model.pl \
      -model     t2s
      -config    ../work/config/NiuTrans.syntax.t2s.user.config \
      -test      ../sample-data/sample-submission-version/Test-set/Niu.test.tree.txt \
      -output    ../work/syntax.trans.result/Niu.test.syntax.t2s.translated.en.txt
```

[9] you can still modify user config file before testing

Command (tree-to-tree)

```
$ cd NiuTrans/scripts/
$ mkdir ../work/syntax.trans.result/ -p
$ perl NiuTrans-syntax-decoder-model.pl \
      -model     t2t
      -config    ../work/config/NiuTrans.syntax.t2t.user.config \
      -test      ../sample-data/sample-submission-version/Test-set/Niu.test.tree.txt \
      -output    ../work/syntax.trans.result/Niu.test.syntax.t2t.translated.en.txt
```

where

`-model` specifies what type of rules can be used to mert, its value can be "s2t", "t2s" or "t2t".

`-config` specifies the configuration file.

`-test` specifies the test dataset (one sentence per line).

`-output` specifies the translation result file (the result is dumped to "stdout" if this option is not specified).

Output: the (1-best) translation file for different models in "/NiuTrans/work/syntax.trans.result".

Output (s2t translation result in NiuTrans/work/syntax.trans.result)

```
- Niu.test.syntax.s2t.translated.en.txt   ▷ 1-best translation of the test sentences
```

Output (t2s translation result in NiuTrans/work/syntax.trans.result)

```
- Niu.test.syntax.t2s.translated.en.txt   ▷ 1-best translation of the test sentences
```

Output (t2t translation result in NiuTrans/work/syntax.trans.result)

```
- Niu.test.syntax.t2t.translated.en.txt   ▷ 1-best translation of the test sentences
```

Chapter 5

Additional Features

In this section several useful features and tips are described. Check them out!

5.1 Generating N-Best Lists

It is trivial to generate n-best translations using *NiuTrans*. What you need is simply setting parameter "nbest" defined in "NiuTrans.phrase.user.config". E.g. if you want to generate a list of 50-best translations, you can modify "NiuTrans.phrase.user.config" as follows:

NiuTrans.phrase.user.config

```
...
# how many translations are dumped
param="nbest"                    value="50"
...
```

5.2 Enlarging Beam Width

As beam search is involved in the decoding process, *NiuTrans* offers a parameter to control the maximum number of hypotheses that are kept in each search step (i.e., *beam size* or *beam width*). When a larger beam width is adopted, there would be fewer search errors and (probably) higher translation accuracy. On the other hand, if a higher decoding speed is required, it is suggested to choose a small beam width. In *NiuTrans*, beam width is controlled by the parameter "beamsize" defined in "NiuTrans.phrase.user.config". E.g. if you wish to choose a beam of width 100, you can modify "NiuTrans.phrase.user.config", as follows:

NiuTrans.phrase.user.config

```
...
# beam size (or beam width)
param="beamsize"                    value="100"
...
```

5.3 Supported Pruning Methods

In addition to beam search (or *beam pruning*), the current version of *NiuTrans* supports two pruning methods: punctuation pruning and cube pruning. In punctuation pruning, the input sentence is first divided into smaller segments according to punctuations (such as commas). The decoding is then performed on each segment individually. The translation of the whole is generated by gluing the translations of these segments. The second method can be regarded as an instance of heuristic search. Here we re-implement the method described in (Chiang, 2007).

To activate the two pruning techniques, users can fire triggers "usepuncpruning" and "usecubepruning" defined in "NiuTrans.phrase.user.config". Of course, each of them can be used individually.

NiuTrans.phrase.user.config

```
...
# use punctuation pruning (1) or not (0)
param="usepuncpruning"              value="1"

# use cube-pruning (1) or not (0)
param="usecubepruning"              value="1"
...
```

5.4 Speeding up the Decoder

To speed-up decoding, a straightforward solution is to prune the space using various (aggressive pruning methods). As described above, punctuation pruning and cube pruning can be employed for system speed-up. By default both of them are activated in the *NiuTrans* system (On Chinese-English translation tasks, they generally lead to a 10-fold speed improvement). Another way for system speed-up is to run the system in multi-thread mode when more than one CPU/core are available. To execute the decoding process on multiple threads, users can use the parameter "nthread" defined in "NiuTrans.phrase.user.config". E.g. if you want to run decoder with 6 threads, you can set "nthread" like this

NiuTrans.phrase.user.config

```
...
# number of threads
param="nthread"                    value="6"
 ...
```

As several (very large) tables or model files are required for running the *NiuTrans* system (See the config file), it is also time consuming to load them before we start the "real" decoding process. To reduce the time of loading various resource files, a straightforward solution is to filter the translation table and reordering model with input sentences, or load those phrase translations as needed during decoding. These features will be supported in the later version of the system.

5.5 Involving More Reference Translations

The *NiuTrans* system does not has any upper limit on the number of reference translations used in either weight tuning or evaluation. E.g. if you want to use three reference translations for weight tuning, you can format your tuning data file as follows (Note that "#" indicates a comment here, and SHOULD NOT appear in users' file).

Sample file (Tuning set)

```
aozhou chongxin kaifang zhu manila dashiguan        # sentence-1
                                                    # a blank line
australia reopens embassy in manila                 # the 1st reference translation
australia reopened manila embassy                   # the 2nd reference translation
australia reopens its embassy to manila             # the 3rd reference translation
aozhou shi yu beihan youbangjiao ...                # sentence-2
```

Then set the `-nref` accordingly. Take the phrase-based engine for instance. For weight tuning, you need run the script as follows (Note: `-nref 3`):

Command

```
$ perl NiuTrans-phrase-mert-model.pl \
      -dev    ../sample-data/sample-submission-version/Dev-set/Niu.dev.txt \
      -c      ../work/NiuTrans.phrase.user.config \
      -nref   1 \
      -r      3 \
      -l      ../work/mert-model.log
```

For evaluation (Note: `-nref 3`),

Command

```
...
$ perl NiuTrans-generate-xml-for-mteval.pl \
      -1f      1best.out \
      -tf      test-ref.txt \
      -rnum    3
...
```

5.6 Using Higher Order N-gram Language Models

Generally a higher order of language model is helpful for most translations tasks. In *NiuTrans*, users can easily build and decode with higher order of language models. First, you need to specify the order for n-gram language model in the LM training step. E.g. if you prefers a 5-gram language model, you can type the following command to train the LM (where `-n 5` means the order of LM is 5)

Command

```
$ ../bin/NiuTrans.LMTrainer \
      -t    sample-submission-version/LM-training-set/e.lm.txt \
      -n    5 \
      -v    lm.vocab \
      -m    lm.trie.data
```

Then set the decoding config file accordingly (where `-ngram 5` means the order of LM is 5)

Command

```
$ cd scripts/
$ perl NiuTrans-phrase-generate-mert-config.pl \
      -tmdir    ../work/model/ \
      -lmdir    ../work/lm/ \
      -ngram    5 \
      -o        ../work/NiuTrans.phrase.user.config
```

5.7 Controlling Phrase Table Size

To avoid extremely large phrase tables, "/config/NiuTrans.phrase.train.model.config" defines two parameters `Max-Source-Phrase-Size` and `Max-Target-Phrase-Size` which control the maximum numbers of words on source-side and target-side of a phrase-pair, respectively. Generally, both the two parameters

greatly impact the number of extracted phrase-pairs. Note that, although extracting larger phrases can increase the coverage rate of a phrase table, it does not always benefit the BLEU improvement due to the data sparseness problem.

Another way to reduce the size of phrase table is to throw away the low-frequency phrases. This can be done using the parameter `Phrase-Cut-Off` defined in `/config/NiuTrans.phrase.train.model.config`. When `Phrase-Cut-Off` is set to n, all phrases appearing equal to or less than n times are thrown away.

E.g. the following example shows how to obtain a phrase table with reasonable size. In this setting, the maximum number of source words and target words are set to 3 and 5, respectively. Moreover, all phrases with frequency 1 are filtered out.

NiuTrans.phrase.user.config

```
...
param="Max-Source-Phrase-Size"              value="3"
param="Max-Target-Phrase-Size"              value="5"
param="Phrase-Cut-Off"                      value="1"
...
```

5.8 Scaling ME-based Reordering Model to Larger Corpus

In general, the size of the (ME-based) reordering model increases dramatically as more training data is involved. *NiuTrans* offers several parameters to control the size of resulting model. They are defined in the configuration file "/config/NiuTrans.phrase.train.model.config", and start with suffix "ME-".

- `ME-max-src-phrase-len` and `ME-max-tar-phrase-len` control the maximum numbers of words appearing in source-side phrase and target-side phrase. Obviously larger `ME-max-src-phrase-len` (or `ME-max-tar-phrase-len`) means a smaller model file.
- `ME-null-algn-word-num` controls the number of unaligned target words that appear between two adjacent blocks.
- `ME-use-src-parse-pruning` is a trigger which indicates whether source-side parse is used to guide the training sample extraction. In our in-house experiments, using source-side parse as constraints can greatly reduce the size of resulting model but does not lose BLEU score significantly.
- `ME-src-parse-path` specifies the file of source parses (one parse per line). It is meaningful only when `ME-use-src-parse-pruning` is turned on.
- `ME-max-sample-num` controls the maximum number of extracted samples for training the ME model. Because the ME trainer (`maxent`) cannot work on a very large training data-set, controlling the maximum number of extracted (training) samples is a reasonable way to avoid the unacceptable training time and memory cost. By default, `ME-max-sample-num` is set to 5000000 in the NiuTrans

system. This setting means that the system only considers the first 5,000,000 samples in model training.

To train the ME-based reordering model on a larger data set, it is recommended to set the above parameters as follows. Note that it requires users to provide the source-side parse trees (See `ME-use-src-parse-pruning` and `ME-src-parse-path`).

NiuTrans.phrase.train.model.config (Settings of ME-based Reordering Model)

```
param="ME-max-src-phrase-len"      value="3"
param="ME-max-tar-phrase-len"      value="5"
param="ME-null-algn-word-num"      value="1"
param="ME-use-src-parse-pruning"   value="1"    # if you have source parses
param="ME-src-parse-path"          value="/path/to/src-parse/"
param="ME-max-sample-num"          value="-1"   # depends on how large your
                                                # corpus is and can be set to a
                                                # positive number as needed
```

5.9 Scaling MSD Reordering Model to Larger Corpus

It is worth pointing out that the *NiuTrans* system have three models to calculate the probabilities of the three reordering types (M, S, D). Users can choose one of them with the parameter "MSD-model-type". When "MSD-model-type" is set to "1", the MSD reordering is modeled on word-level, as what the Moses system does. In addition to the basic model, the phrase-based MSD model and the hierarchical phrase-based MSD model (Galley et al., 2008) are also implemented. They can be activated when "MSD-model-type" is set to "2" or "3".

When trained on a large corpus, the generated MSD model might be very large. The situations even more severe when model "3" (i.e., hierarchical phrase-based MSD model) is involved. To alleviate this problem, users can use the parameter "MSD-filter-method" which filters the MSD model using phrase translation table (any entry that is not covered by the phrase table will be excluded).

Also, users can use the parameter "MSD-max-phrase-len" to limit the maximum number of words in a source or target phrase. This parameter can effectively reduce the size of the generated MSD model.

Below gives an sample config file for creating a MSD model with an acceptable size.

NiuTrans.phrase.train.model.config (Settings of MSD Reordering Model)

```
param="MSD-model-type"         value="1"             # "1", "2" or "3"
param="MSD-filter-method"      value="tran-table"    # "tran-table" or "msd-sum-1"
param="MSD-max-phrase-len"     value="7"             # number greater than 0
```

5.10 Adding Self-developed Features into NiuTrans

The *NiuTrans* system allows users to add self-developed features into the phrase translation table. By default, each entry in the translation table is associated with 6 features. E.g. below is a sample table ("phrase.translation.table"), where each entry is coupled with a 6-dimension feature vector.

Phrase Table in Default Format (phrase.translation.table)

```
...
yiding ||| must ||| -2.35374 -2.90407 -1.60161 -2.12482 1 0
yiding ||| a certain ||| -2.83659 -1.07536 -4.97444 -1.90004 1 0
yiding ||| be ||| -4.0444 -5.74325 -2.32375 -4.46486 1 0
yiding ||| be sure ||| -4.21145 -1.3278 -5.75147 -3.32514 1 0
yiding ||| ’ ll ||| -5.10527 -5.32301 -8.64566 -4.80402 1 0
...
```

To add new features into the table, users can append them to the feature vectors shown above. E.g. suppose that we wish to add a feature that indicates whether the phrase pair appears only once in the training data or not (appears two times or more). We can update the above table, as follows:

Phrase Table with a Newly-added Feature(phrase.translation.table)

```
...
yiding ||| must ||| -2.35374 -2.90407 -1.60161 -2.12482 1 0 0
yiding ||| a certain ||| -2.83659 -1.07536 -4.97444 -1.90004 1 0 0
yiding ||| be ||| -4.0444 -5.74325 -2.32375 -4.46486 1 0 1
yiding ||| be sure ||| -4.21145 -1.3278 -5.75147 -3.32514 1 0 1
yiding ||| ’ ll ||| -5.10527 -5.32301 -8.64566 -4.80402 1 0 1
...
```

We then modify the config file "NiuTrans.phrase.user.config" to activate the newly-introduced feature in the decoder.

Activating the New Feature (NiuTrans.phrase.user.config)

```
param="freefeature"              value="1"
param="tablefeatnum"             value="7"
```

where "`freefeature`" is a trigger that indicates wether the additional features are used or not. "`tablefeatnum`" sets the number of features defined in the table.

5.11 Plugging External Translations into the Decoder

The *NiuTrans* system also defines some special markups to support external translations specified by users. E.g. below is sample sentence to be decoded.

bidetaile shi yiming yingguo zishen jinrong fenxishi .
(Peter Taylor is a senior financial analyst at UK .)

If you have prior knowledge about how to translate "bidetaile" and "yingguo", you can add your own translations into the decoding using some markups. The following is an example:

Using External Translations (dev or test file)

```
bidetaile shi yiming yingguo zishen jinrong fenxishi .  |||| {0 ||| 0 ||| Peter Taylor |||
$ne ||| bidetaile} {3 ||| 3 ||| UK ||| $ne ||| yingguo}
```

where "||||" is a separator, "{0 ||| 0 ||| Peter Taylor ||| $ne ||| bidetaile}" and "{3 ||| 3 ||| UK ||| $ne ||| yingguo}" are two user-defined translations. Each consists of 5 terms. The first two numbers indicate the span to be translated; the third term is the translation specified by users; the fourth term indicates the type of translation; and the last term repeats the corresponding source word sequence.

Chapter 6

Acknowledgements

This project is supported in part by the National Science Foundation of China (60873091; 61073140), Specialized Research Fund for the Doctoral Program of Higher Education (20100042110031), and the Fundamental Research Funds for the Central Universities.

We also would like to thank Wenliang Chen, Jianqing Cui, Ji Ma, Matthias Huck and Kehai Chen for their valuable suggestions for improving NiuTrans and this document.

Appendix A

Data Preparation

Sample Data (NiuTrans/sample-data/sample-submission-version)

```
sample-submission-version/
   TM-training-set/              ▷ word-aligned bilingual corpus
                                 ▷ (100,000 sentence-pairs)
      chinese.txt                ▷ source sentences
      english.txt                ▷ target sentences (case-removed)
      Alignment.txt              ▷ word alignments of the sentence-pairs
   LM-training-set/
      e.lm.txt                   ▷ monolingual corpus for training language model
                                 ▷ (100K target sentences)
   Dev-set/
      Niu.dev.txt                ▷ development dataset for weight tuning
                                 ▷ (400 sentences)
   Test-set/
      Niu.test.txt               ▷ test dataset (1K sentences)
   Reference-for-evaluation/
      Niu.test.reference         ▷ references of the test sentences (1K sentences)
   description-of-the-sample-data ▷ a description of the sample data
```

- The *NiuTrans* system is a "data-driven" MT system which requries "data" for training and/or tuning the system. It requries users to prepare the following data files before running the system.

  1. **Training data**: bilingual sentence-pairs and word alignments.
  2. **Tuning data**: source sentences with one or more reference translations.
  3. **Test data**: some new sentences.

4. **Evaluation data**: reference translations of test sentences.

In the *NiuTrans* package, some sample files are offered for experimenting with the system and studying the format requirement. They are located in "NiuTrans/sample-data/sample-submission-version".

- Format: please unpack "NiuTrans/sample-data/sample.tar.gz", and refer to "description-of-the-sample-data" to find more information about data format.
- In the following, the above data files are used to illustrate how to run the *NiuTrans* system (e.g. how to train MT models, tune feature weights, and decode test sentences).

Appendix B

Brief Usage

B.1 Brief Usage for NiuTrans.Phrase

Please jump to **Chapter 2 Quick Walkthrough** for more detail.

B.2 Brief Usage for NiuTrans.Hierarchy

B.2.1 Obtaining Hierarchy Rules

- **Instructions** (perl is required. Also, Cygwin is required for Windows users)

Command

```
$ cd NiuTrans/sample-data/
$ tar xzf sample.tar.gz
$ cd ../
$ mkdir work/model.hierarchy/ -p
$ cd scripts/
$ perl NiuTrans-hierarchy-train-model.pl \
      -src   ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tgt   ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -aln   ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -out   ../work/model.hierarchy/hierarchy.rule.table
```

where

`-out` specifies the generated hierarchy rule table.

`-src`, `-tgt` and `-aln` specify the source sentences, the target sentences and the alignments between them (one sentence per line).

- **Output**: one file are generated and placed in "NiuTrans/work/model.hierarchy/":

Output (NiuTrans/work/model.hierarchy/)

```
- hierarchy.rule.table        ▷ hierarchy rule table
```

- Note: Please enter the "scripts/" directory before running the script "NiuTrans-hierarchy-train-model.pl".

B.2.2 Training n-gram language model

- **Instructions**

Command

```
$ cd ../
$ mkdir work/lm/
$ cd scripts/
$ perl NiuTrans-training-ngram-LM.pl \
      -corpus    ../sample-data/sample-submission-version/LM-training-set/e.lm.txt \
      -ngram     3 \
      -vocab     ../work/lm/lm.vocab \
      -lmbin     ../work/lm/lm.trie.data
```

where

`-corpus` specifies the training data file (i.e., a collection of target-language sentences. one sentence per line).

`-ngram` specifies the order of n-gram LM. E.g. "`-ngram` 3" indicates a 3-gram language model.

`-vocab` specifies where the target-side vocabulary is generated.

`-lmbin` specifies where the language model file is generated.

- **Output**: two files are generated and placed in "NiuTrans/work/lm/":

Output (NiuTrans/work/lm/)

```
- lm.vocab                ▷ target-side vocabulary
- lm.trie.data            ▷ binary-encoded language model
```

B.2.3 Generating Configuration File

- **Instructions**

Command

```
$ cd scripts/
$ perl NiuTrans-hierarchy-generate-mert-config.pl \
      -rule    ../work/model.hierarchy/hierarchy.rule.table \
      -lmdir   ../work/lm/ \
      -nref    1 \
      -ngram   3 \
      -out     ../work/NiuTrans.hierarchy.user.config
```

where

`-rule` specifies the hierarchy rule table.

`-lmdir` specifies the directory that holds the n-gram language model and the target-side vocabulary.

`-nref` specifies how many reference translations per source-sentence are provided.

`-ngram` specifies the order of n-gram language model.

`-out` specifies the output (i.e. a config file).

- **Output**: a config file is generated and placed in "NiuTrans/work/".

Output (NiuTrans/work/)

```
- NiuTrans.hierarchy.user.config        ▷ configuration file for MERT and decoding
```

B.2.4 Weight Tuning

- **Instructions**

Command

```
$ perl NiuTrans-hierarchy-mert-model.pl \
      -config    ../work/NiuTrans.hierarchy.user.config \
      -dev       ../sample-data/sample-submission-version/Dev-set/Niu.dev.txt \
      -nref      1 \
      -round     3 \
      -log       ../work/mert-model.log
```

where

`-config` specifies the configuration file generated in the previous steps.

`-dev` specifies the development dataset (or tuning set) for weight tuning.

`-nref` specifies how many reference translations per source-sentence are provided

`-round` specifies how many rounds the MERT performs (by default, 1 round = 15 MERT iterations).

`-log` specifies the log file generated by MERT.

- **Output**: the optimized feature weights are recorded in the configuration file "NiuTrans/work/NiuTrans.hierarchy.user.config". They will then be used in decoding the test sentences.

B.2.5 Decoding Test Sentences

- **Instructions**

Command

```
$ perl NiuTrans-hierarchy-decoder-model.pl \
      -config    ../work/NiuTrans.hierarchy.user.config \
      -test      ../sample-data/sample-submission-version/Test-set/Niu.test.txt \
      -output    1best.out
```

where

`-config` specifies the configuration file.

`-test` specifies the test dataset (one sentence per line).

`-output` specifies the translation result file (the result is dumped to "stdout" if this option is not specified).

- **Output**: a new file is generated in "NiuTrans/scripts/":

Output (NiuTrans/scripts/)

```
- 1best.out            ▷ 1-best translation of the test sentences
```

B.2.6 Evaluation

- **Instructions**

Command

```
$ perl NiuTrans-generate-xml-for-mteval.pl \
      -1f    1best.out \
      -tf    ../sample-data/sample-submission-version/Reference-for-evaluation/Niu.test.reference \
      -rnum  1
$ perl mteval-v13a.pl -r ref.xml -s src.xml -t tst.xml
```

where

`-1f` specifies the file of the 1-best translations of the test dataset.

`-tf` specifies the file of the source sentences and their reference translations of the test dataset.

`-rnum` specifies how many reference translations per test sentence are provided.

`-r` specifies the file of the reference translations.

`-s` specifies the file of source sentence.

`-t` specifies the file of (1-best) translations generated by the MT system.

- **Output**: The IBM-version BLEU score is displayed on the screen.
- **Note**: script mteval-v13a.pl replies on the package XML::Parser. If XML::Parser is not installed on your system, please follow the following commands to install it.

Command

```
$ su root
$ tar xzf XML-Parser-2.41.tar.gz
$ cd XML-Parser-2.41/
$ perl Makefile.PL
$ make install
```

B.3 Brief Usage for NiuTrans.Syntax - string to tree

B.3.1 Obtaining Syntax Rules

- **Instructions** (perl is required. Also, Cygwin is required for Windows users)

Command

```
$ cd NiuTrans/sample-data/
$ tar xzf sample.tar.gz
$ cd ../
$ mkdir work/model.syntax.s2t/ -p
$ cd scripts/
$ perl NiuTrans-syntax-train-model.pl \
      -model   s2t \
      -src     ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tgt     ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -aln     ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -ttree   ../sample-data/sample-submission-version/TM-training-set/english.tree.txt \
      -out     ../work/model.syntax.s2t/syntax.string2tree.rule
```

where

`-model` specifies SMT translation model, the model decides what type of rules can be generated, its value can be "s2t", "t2s" or "t2t", default "t2s". For string-to-tree model, the value is "s2t".

`-src`, `-tgt` and `-aln` specify the source sentences, the target sentences and the alignments between them (one sentence per line).

`-ttree` specifies path to the target sentence parse tree file, The parse tree format is like Berkeley Parser's output.

`-out` specifies the generated string-to-tree syntax rule table.

- **Output**: three files are generated and placed in "NiuTrans/work/model.syntax.s2t/".

Output (NiuTrans/work/model.syntax.s2t/)

```
- syntax.string2tree.rule          ▷ syntax rule table
- syntax.string2tree.rule.bina     ▷ binarization rule table for decoder
- syntax.string2tree.rule.unbina   ▷ unbinarization rule table for decoder
```

- **Note**: Please enter the "NiuTrans/scripts/" directory before running the script "NiuTrans-syntax-train-model.pl".

B.3.2 Training n-gram language model

- **Instructions**

Command

```
$ cd ../
$ mkdir work/lm/
$ cd scripts/
$ perl NiuTrans-training-ngram-LM.pl \
      -corpus    ../sample-data/sample-submission-version/LM-training-set/e.lm.txt \
      -ngram     3 \
      -vocab     ../work/lm/lm.vocab \
      -lmbin     ../work/lm/lm.trie.data
```

where

`-corpus` specifies the training data file (i.e., a collection of target-language sentences. one sentence per line).

`-ngram` specifies the order of n-gram LM. E.g. "`-ngram` 3" indicates a 3-gram language model.

`-vocab` specifies where the target-side vocabulary is generated.

`-lmbin` specifies where the language model file is generated.

- **Output**: two files are generated and placed in "NiuTrans/work/lm/".

Output (NiuTrans/work/lm/)

```
- lm.vocab                ▷ target-side vocabulary
- lm.trie.data            ▷ binary-encoded language model
```

B.3.3 Generating Configuration File

- **Instructions**

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/config/ -p
$ perl NiuTrans-syntax-generate-mert-config.pl \
      -model          s2t \
      -syntaxrule     ../work/model.syntax.s2t/syntax.string2tree.rule.bina \
      -lmdir          ../work/lm/ \
      -nref           1 \
      -ngram          3 \
      -out            ../work/config/NiuTrans.syntax.s2t.user.config
```

where

`-model` specifies what type of rules can be used to mert, its value can be "s2t", "t2s" or "t2t".

`-syntaxrule` specifies the syntax-rule table.

`-lmdir` specifies the directory that holds the n-gram language model and the target-side vocabulary.

`-nref` specifies how many reference translations per source-sentence are provided.

`-ngram` specifies the order of n-gram language model.

`-out` specifies the output (i.e. a config file).

- **Output**: a config file is generated and placed in "NiuTrans/work/config/". Users can modify this generated config file as needed.

Output (NiuTrans/work/config/)

```
- NiuTrans.syntax.s2t.user.config        ▷ configuration file for MERT and decoding
```

B.3.4 Weight Tuning

- **Instructions**

Command

```
$ cd NiuTrans/scripts/
$ perl NiuTrans-syntax-mert-model.pl \
      -model     s2t \
      -config    ../work/config/NiuTrans.syntax.s2t.user.config \
      -dev       ../sample-data/sample-submission-version/Dev-set/Niu.dev.tree.txt \
      -nref      1 \
      -round     3 \
      -log       ../work/syntax-s2t-mert-model.log
```

where

`-model` specifies what type of rules can be used to mert, its value can be "s2t", "t2s" or "t2t".

`-config` specifies the configuration file generated in the previous steps.

`-dev` specifies the development dataset (or tuning set) for weight tuning.

`-nref` specifies how many reference translations per source-sentence are provided.

`-round` specifies how many rounds the MERT performs (by default, 1 round = 15 MERT iterations).

`-log` specifies the log file generated by MERT.

- **Output**: the optimized feature weights are recorded in the configuration file "NiuTrans/work/config/NiuTrans.syntax.s2t.user.config". They will then be used in decoding the test sentences.

B.3.5 Decoding Test Sentences

- **Instructions**

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/syntax.trans.result/ -p
$ perl NiuTrans-syntax-decoder-model.pl \
      -model     s2t
      -config    ../work/config/NiuTrans.syntax.s2t.user.config \
      -test      ../sample-data/sample-submission-version/Test-set/Niu.test.tree.txt \
      -output    ../work/syntax.trans.result/Niu.test.syntax.s2t.translated.en.txt
```

where

`-model` specifies what type of rules can be used to mert, its value can be "s2t", "t2s" or "t2t".

`-config` specifies the configuration file.

`-test` specifies the test dataset (one sentence per line).

`-output` specifies the translation result file (the result is dumped to "stdout" if this option is not specified).

- **Output**: the (1-best) translations in "/NiuTrans/work/syntax.trans.result".

Output (NiuTrans/work/syntax.trans.result)

```
- Niu.test.syntax.s2t.translated.en.txt   ▷ 1-best translation of the test sentences
```

B.3.6 Evaluation

- **Instructions**

Command

```
$ perl NiuTrans-generate-xml-for-mteval.pl \
      -1f    ../work/syntax.trans.result/Niu.test.syntax.s2t.translated.en.txt \
      -tf    ../sample-data/sample-submission-version/Reference-for-evaluation/Niu.test.reference \
      -rnum  1
$ perl mteval-v13a.pl \
      -r     ref.xml \
      -s     src.xml \
      -t     tst.xml
```

where

`-1f` specifies the file of the 1-best translations of the test dataset.

`-tf` specifies the file of the source sentences and their reference translations of the test dataset.

`-rnum` specifies how many reference translations per test sentence are provided.

`-r` specifies the file of the reference translations.

`-s` specifies the file of source sentence.

`-t` specifies the file of (1-best) translations generated by the MT system.

- **Output**: The IBM-version BLEU score is displayed on the screen.
- **Note**: script mteval-v13a.pl replies on the package XML::Parser. If XML::Parser is not installed on your system, please follow the following commands to install it.

Command

```
$ su root
$ tar xzf XML-Parser-2.41.tar.gz
$ cd XML-Parser-2.41/
$ perl Makefile.PL
$ make install
```

B.4 Brief Usage for NiuTrans.Syntax - tree to string

B.4.1 Obtaining Syntax Rules

- **Instructions** (perl is required. Also, Cygwin is required for Windows users)

Command

```
$ cd NiuTrans/sample-data/
$ tar xzf sample.tar.gz
$ cd ../
$ mkdir work/model.syntax.t2s/ -p
$ cd scripts/
$ perl NiuTrans-syntax-train-model.pl \
      -model   t2s \
      -src     ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tgt     ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -aln     ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -stree   ../sample-data/sample-submission-version/TM-training-set/chinese.tree.txt \
      -out     ../work/model.syntax.t2s/syntax.tree2string.rule
```

where

`-model` specifies SMT translation model, the model decides what type of rules can be generated, its value can be "s2t", "t2s" or "t2t", default "t2s". For tree-to-string model, the value is "t2s".

`-src`, `-tgt` and `-aln` specify the source sentences, the target sentences and the alignments between them (one sentence per line).

`-stree` specifies path to the source sentence parse tree file, The parse tree format is like Berkeley Parser's output.

`-out` specifies the generated tree-to-string syntax rule table.

- **Output**: three files are generated and placed in "NiuTrans/work/model.syntax.t2s/".

Output (NiuTrans/work/model.syntax.t2s/)

```
- syntax.tree2string.rule          ▷ syntax rule table
- syntax.tree2string.rule.bina     ▷ binarization rule table for decoder
- syntax.tree2string.rule.unbina   ▷ unbinarization rule table for decoder
```

- **Note**: Please enter the "NiuTrans/scripts/" directory before running the script "NiuTrans-syntax-train-model.pl".

B.4.2 Training n-gram language model

- **Instructions**

Command

```
$ cd ../
$ mkdir work/lm/
$ cd scripts/
$ perl NiuTrans-training-ngram-LM.pl \
      -corpus    ../sample-data/sample-submission-version/LM-training-set/e.lm.txt \
      -ngram     3 \
      -vocab     ../work/lm/lm.vocab \
      -lmbin     ../work/lm/lm.trie.data
```

where

`-corpus` specifies the training data file (i.e., a collection of target-language sentences. one sentence per line).

`-ngram` specifies the order of n-gram LM. E.g. "`-n` 3" indicates a 3-gram language model.

`-vocab` specifies where the target-side vocabulary is generated.

`-lmbin` specifies where the language model file is generated.

- **Output**: two files are generated and placed in "NiuTrans/work/lm/".

Output (NiuTrans/work/lm/)

```
- lm.vocab                ▷ target-side vocabulary
- lm.trie.data            ▷ binary-encoded language model
```

B.4.3 Generating Configuration File

- **Instructions**

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/config/ -p
$ perl NiuTrans-syntax-generate-mert-config.pl \
      -model          t2s \
      -syntaxrule     ../work/model.syntax.t2s/syntax.tree2string.rule.bina \
      -lmdir          ../work/lm/ \
      -nref           1 \
      -ngram          3 \
      -out            ../work/config/NiuTrans.syntax.t2s.user.config
```

where

`-model` specifies what type of rules can be used to mert, its value can be "s2t", "t2s" or "t2t".

`-syntaxrule` specifies the syntax-rule table.

`-lmdir` specifies the directory that holds the n-gram language model and the target-side vocabulary.

`-nref` specifies how many reference translations per source-sentence are provided.

`-ngram` specifies the order of n-gram language model.

`-out` specifies the output (i.e. a config file).

- **Output**: a config file is generated and placed in "NiuTrans/work/config/". Users can modify this generated config file as needed.

Output (NiuTrans/work/config/)

```
- NiuTrans.syntax.t2s.user.config        ▷ configuration file for MERT and decoding
```

B.4.4 Weight Tuning

- **Instructions**

Command

```
$ cd NiuTrans/scripts/
$ perl NiuTrans-syntax-mert-model.pl \
      -model     t2s \
      -config    ../work/config/NiuTrans.syntax.t2s.user.config \
      -dev       ../sample-data/sample-submission-version/Dev-set/Niu.dev.tree.txt \
      -nref      1 \
      -round     3 \
      -log       ../work/syntax-t2s-mert-model.log
```

where

`-model` specifies what type of rules can be used to mert, its value can be "s2t", "t2s" or "t2t".

`-config` specifies the configuration file generated in the previous steps.

`-dev` specifies the development dataset (or tuning set) for weight tuning.

`-nref` specifies how many reference translations per source-sentence are provided.

`-round` specifies how many rounds the MERT performs (by default, 1 round = 15 MERT iterations).

`-log` specifies the log file generated by MERT.

- **Output**: the optimized feature weights are recorded in the configuration file "NiuTrans/work/config/NiuTrans.syntax.t2s.user.config". They will then be used in decoding the test sentences.

B.4.5 Decoding Test Sentences

- **Instructions**

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/syntax.trans.result/ -p
$ perl NiuTrans-syntax-decoder-model.pl \
      -model     t2s
      -config    ../work/config/NiuTrans.syntax.t2s.user.config \
      -test      ../sample-data/sample-submission-version/Test-set/Niu.test.tree.txt \
      -output    ../work/syntax.trans.result/Niu.test.syntax.t2s.translated.en.txt
```

where

`-model` specifies what type of rules can be used to mert, its value can be "s2t", "t2s" or "t2t".

`-config` specifies the configuration file.

`-test` specifies the test dataset (one sentence per line).

`-output` specifies the translation result file (the result is dumped to "stdout" if this option is not specified).

- **Output**: the (1-best) translations in "/NiuTrans/work/syntax.trans.result".

Output (NiuTrans/work/syntax.trans.result)

```
- Niu.test.syntax.t2s.translated.en.txt   ▷ 1-best translation of the test sentences
```

B.4.6 Evaluation

- **Instructions**

Command

```
$ perl NiuTrans-generate-xml-for-mteval.pl \
      -1f    ../work/syntax.trans.result/Niu.test.syntax.t2s.translated.en.txt \
      -tf    ../sample-data/sample-submission-version/Reference-for-evaluation/Niu.test.reference \
      -rnum  1
$ perl mteval-v13a.pl \
      -r     ref.xml \
      -s     src.xml \
      -t     tst.xml
```

where

`-1f` specifies the file of the 1-best translations of the test dataset.

`-tf` specifies the file of the source sentences and their reference translations of the test dataset.

`-rnum` specifies how many reference translations per test sentence are provided.

`-r` specifies the file of the reference translations.

`-s` specifies the file of source sentence.

`-t` specifies the file of (1-best) translations generated by the MT system.

- **Output**: The IBM-version BLEU score is displayed on the screen.
- **Note**: script mteval-v13a.pl replies on the package XML::Parser. If XML::Parser is not installed on your system, please follow the following commands to install it.

Command

```
$ su root
$ tar xzf XML-Parser-2.41.tar.gz
$ cd XML-Parser-2.41/
$ perl Makefile.PL
$ make install
```

B.5 Brief Usage for NiuTrans.Syntax - tree to tree

B.5.1 Obtaining Syntax Rules

- **Instructions** (perl is required. Also, Cygwin is required for Windows users)

Command

```
$ cd NiuTrans/sample-data/
$ tar xzf sample.tar.gz
$ cd ../
$ mkdir work/model.syntax.t2t/ -p
$ cd scripts/
$ perl NiuTrans-syntax-train-model.pl \
      -model   t2t \
      -src     ../sample-data/sample-submission-version/TM-training-set/chinese.txt \
      -tgt     ../sample-data/sample-submission-version/TM-training-set/english.txt \
      -aln     ../sample-data/sample-submission-version/TM-training-set/Alignment.txt \
      -stree   ../sample-data/sample-submission-version/TM-training-set/chinese.tree.txt \
      -ttree   ../sample-data/sample-submission-version/TM-training-set/english.tree.txt \
      -out     ../work/model.syntax.t2t/syntax.tree2tree.rule
```

where

`-model` specifies SMT translation model, the model decides what type of rules can be generated, its value can be "s2t", "t2s" or "t2t", default "t2s". For tree-to-tree model, the value is "t2t".

`-src`, `-tgt` and `-aln` specify the source sentences, the target sentences and the alignments between them (one sentence per line).

`-stree` specifies path to the source sentence parse tree file, The parse tree format is like Berkeley Parser's output.

`-ttree` specifies path to the target sentence parse tree file, The parse tree format is like Berkeley Parser's output.

`-out` specifies the generated tree-to-tree syntax rule table.

- **Output**: three files are generated and placed in "NiuTrans/work/model.syntax.t2t/".

Output (NiuTrans/work/model.syntax.t2t/)

```
- syntax.tree2tree.rule          ▷ syntax rule table
- syntax.tree2tree.rule.bina     ▷ binarization rule table for decoder
- syntax.tree2tree.rule.unbina   ▷ unbinarization rule table for decoder
```

- **Note**: Please enter the "NiuTrans/scripts/" directory before running the script "NiuTrans-syntax-train-model.pl".

B.5.2 Training n-gram language model

- **Instructions**

Command

```
$ cd ../
$ mkdir work/lm/
$ cd scripts/
$ perl NiuTrans-training-ngram-LM.pl \
      -corpus    ../sample-data/sample-submission-version/LM-training-set/e.lm.txt \
      -ngram     3 \
      -vocab     ../work/lm/lm.vocab \
      -lmbin     ../work/lm/lm.trie.data
```

where

`-corpus` specifies the training data file (i.e., a collection of target-language sentences. one sentence per line).

`-ngram` specifies the order of n-gram LM. E.g. "`-ngram` 3" indicates a 3-gram language model.

`-vocab` specifies where the target-side vocabulary is generated.

`-lmbin` specifies where the language model file is generated.

- **Output**: two files are generated and placed in "NiuTrans/work/lm/".

Output (NiuTrans/work/lm/)

```
- lm.vocab                ▷ target-side vocabulary
- lm.trie.data            ▷ binary-encoded language model
```

B.5.3 Generating Configuration File

- **Instructions**

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/config/ -p
$ perl NiuTrans-syntax-generate-mert-config.pl \
      -model          t2t \
      -syntaxrule     ../work/model.syntax.t2t/syntax.tree2tree.rule.bina \
      -lmdir          ../work/lm/ \
      -nref           1 \
      -ngram          3 \
      -out            ../work/config/NiuTrans.syntax.t2t.user.config
```

where

`-model` specifies what type of rules can be used to mert, its value can be "s2t", "t2s" or "t2t".

`-syntaxrule` specifies the syntax-rule table.

`-lmdir` specifies the directory that holds the n-gram language model and the target-side vocabulary.

`-nref` specifies how many reference translations per source-sentence are provided.

`-ngram` specifies the order of n-gram language model.

`-out` specifies the output (i.e. a config file).

- **Output**: a config file is generated and placed in "NiuTrans/work/config/". Users can modify this generated config file as needed.

Output (NiuTrans/work/config/)

```
- NiuTrans.syntax.t2t.user.config        ▷ configuration file for MERT and decoding
```

B.5.4 Weight Tuning

- **Instructions**

Command

```
$ cd NiuTrans/scripts/
$ perl NiuTrans-syntax-mert-model.pl \
      -model     t2t \
      -config    ../work/config/NiuTrans.syntax.t2t.user.config \
      -dev       ../sample-data/sample-submission-version/Dev-set/Niu.dev.tree.txt \
      -nref      1 \
      -round     3 \
      -log       ../work/syntax-t2t-mert-model.log
```

where

`-model` specifies what type of rules can be used to mert, its value can be "s2t", "t2s" or "t2t".

`-config` specifies the configuration file generated in the previous steps.

`-dev` specifies the development dataset (or tuning set) for weight tuning.

`-nref` specifies how many reference translations per source-sentence are provided.

`-round` specifies how many rounds the MERT performs (by default, 1 round = 15 MERT iterations).

`-log` specifies the log file generated by MERT.

- **Output**: the optimized feature weights are recorded in the configuration file "NiuTrans/work/config/NiuTrans.syntax.t2t.user.config". They will then be used in decoding the test sentences.

B.5.5 Decoding Test Sentences

- **Instructions**

Command

```
$ cd NiuTrans/scripts/
$ mkdir ../work/syntax.trans.result/ -p
$ perl NiuTrans-syntax-decoder-model.pl \
      -model     t2t
      -config    ../work/config/NiuTrans.syntax.t2t.user.config \
      -test      ../sample-data/sample-submission-version/Test-set/Niu.test.tree.txt \
      -output    ../work/syntax.trans.result/Niu.test.syntax.t2t.translated.en.txt
```

where

`-model` specifies what type of rules can be used to mert, its value can be "s2t", "t2s" or "t2t".

`-config` specifies the configuration file.

`-test` specifies the test dataset (one sentence per line).

`-output` specifies the translation result file (the result is dumped to "stdout" if this option is not specified).

- **Output**: the (1-best) translations in "/NiuTrans/work/syntax.trans.result".

Output (NiuTrans/work/syntax.trans.result)

```
- Niu.test.syntax.t2t.translated.en.txt   ▷ 1-best translation of the test sentences
```

B.5.6 Evaluation

- **Instructions**

Command

```
$ perl NiuTrans-generate-xml-for-mteval.pl \
      -1f    ../work/syntax.trans.result/Niu.test.syntax.t2t.translated.en.txt \
      -tf    ../sample-data/sample-submission-version/Reference-for-evaluation/Niu.test.reference \
      -rnum  1
$ perl mteval-v13a.pl \
      -r     ref.xml \
      -s     src.xml \
      -t     tst.xml
```

where

`-1f` specifies the file of the 1-best translations of the test dataset.

`-tf` specifies the file of the source sentences and their reference translations of the test dataset.

`-rnum` specifies how many reference translations per test sentence are provided.

`-r` specifies the file of the reference translations.

`-s` specifies the file of source sentence.

`-t` specifies the file of (1-best) translations generated by the MT system.

- **Output**: The IBM-version BLEU score is displayed on the screen.
- **Note**: script mteval-v13a.pl replies on the package XML::Parser. If XML::Parser is not installed on your system, please follow the following commands to install it.

Command

```
$ su root
$ tar xzf XML-Parser-2.41.tar.gz
$ cd XML-Parser-2.41/
$ perl Makefile.PL
$ make install
```

Bibliography

Alfred V. Aho and Jeffrey D. Ullman. Syntax directed translations and the pushdown assembler. *Journal of Computer and System Sciences*, 3:37–57, 1969.

Hiyan Alshawi, Srinivas Bangalore, and Shona Douglas. Learning dependency translation models as collections of finite state head transducers. *Computational Linguistics*, 26:45–60, 2000.

Adam L. Bergert, Vincent J. Della Pietra, and Stephen A. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22:39–71, 1996.

Peter E. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19:263–311, 1993.

Daniel Cer, Michel Galley, Daniel Jurafsky, and Christopher D. Manning. Phrasal: A statistical machine translation toolkit for exploring new model features. In *Proceedings of the NAACL HLT 2010 Demonstration Session*, pages 9–12, Los Angeles, California, June 2010. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N10-2003`.

David Chiang. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 263–270, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics. doi: 10.3115/1219840.1219873. URL `http://www.aclweb.org/anthology/P05-1033`.

David Chiang. Hierarchical phrase-based translation. *Computational Linguistics*, 33:45–60, 2007.

David Chiang and Kevin Knight. An introduction to synchronous grammars. In *Proceedings of the 44rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*. Association for Computational Linguistics, 2006.

David Chiang, Steve DeNeefe, Yee Seng Chan, and Hwee Tou Ng. Decomposability of translation metrics for improved evaluation and efficient algorithms. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 610–619, Honolulu, Hawaii, October 2008. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D08-1064`.

John DeNero, Shankar Kumar, Ciprian Chelba, and Franz Och. Model combination for machine translation. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 975–983, Los Angeles, California, June 2010. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N10-1141`.

Chris Dyer, Adam Lopez, Juri Ganitkevitch, Jonathan Weese, Ferhan Ture, Phil Blunsom, Hendra Setiawan, Vladimir Eidelman, and Philip Resnik. cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models. In *Proceedings of the ACL 2010 System Demonstrations*, pages 7–12, Uppsala, Sweden, July 2010. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P10-4002`.

Jason Eisner. Learning non-isomorphic tree mappings for machine translation. In *The Companion Volume to the Proceedings of 41st Annual Meeting of the Association for Computational Linguistics*, pages 205–208, Sapporo, Japan, July 2003. Association for Computational Linguistics. doi: 10.3115/1075178.1075217. URL `http://www.aclweb.org/anthology/P03-2039`.

Michel Galley and Christopher D. Manning. A simple and effective hierarchical phrase reordering model. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 848–856, Honolulu, Hawaii, October 2008. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D08-1089`.

Michel Galley, Jonathan Graehl, Kevin Knight, Daniel Marcu, Steve DeNeefe, Wei Wang, and Ignacio Thayer. Scalable inference and training of context-rich syntactic translation models. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 961–968, Sydney, Australia, July 2006. Association for Computational Linguistics. doi: 10.3115/1220175.1220296. URL `http://www.aclweb.org/anthology/P06-1121`.

Liang Huang and David Chiang. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 53–64, Vancouver, British Columbia, October 2005. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W05/W05-1506`.

Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, 2010.

Philipp Koehn, Franz Och, and Daniel Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, Edmonton, June 2003.

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, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, pages 177–180, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P07-2045`.

Zhifei Li, Chris Callison-Burch, Chris Dyer, Sanjeev Khudanpur, Lane Schwartz, Wren Thornton, Jonathan Weese, and Omar Zaidan. Joshua: An open source toolkit for parsing-based machine translation. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*, pages 135–139, Athens, Greece, March 2009. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W09-0424`.

Adam Lopez. Statistical machine translation. *ACM Computing Surveys*, 40:1–49, 2008.

Daniel Marcu and Daniel Wong. A phrase-based,joint probability model for statistical machine translation. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 133–139. Association for Computational Linguistics, July 2002. doi: 10.3115/1118693.1118711. URL `http://www.aclweb.org/anthology/W02-1018`.

Haitao Mi, Liang Huang, and Qun Liu. Forest-based translation. In *Proceedings of ACL-08: HLT*, pages 192–199, Columbus, Ohio, June 2008. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P/P08/P08-1023.

Franz Och. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 160–167, Sapporo, Japan, July 2003. Association for Computational Linguistics. doi: 10.3115/1075096.1075117. URL http://www.aclweb.org/anthology/P03-1021.

Franz Och and Hermann Ney. Discriminative training and maximum entropy models for statistical machine translation. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 295–302, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073133. URL http://www.aclweb.org/anthology/P02-1038.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL http://www.aclweb.org/anthology/P02-1040.

Christoph Tillman. A unigram orientation model for statistical machine translation. In Daniel Marcu Susan Dumais and Salim Roukos, editors, *HLT-NAACL 2004: Short Papers*, pages 101–104, Boston, Massachusetts, USA, May 2 - May 7 2004. Association for Computational Linguistics.

David Vilar, Daniel Stein, Matthias Huck, and Hermann Ney. Jane: Open source hierarchical translation, extended with reordering and lexicon models. In *Proceedings of the Joint Fifth Workshop on Statistical Machine Translation and MetricsMATR*, pages 262–270, Uppsala, Sweden, July 2010. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W10-1738.

Dekai Wu. Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. *Computational Linguistics*, 23:377–404, 1997.

Tong Xiao, Rushan Chen, Tianning Li, Muhua Zhu, Jingbo Zhu, Huizhen Wang, and Feiliang Ren. Neutrans: a phrase-based smt system for cwmt2009. In *Proceedings of the 5th China Workshop on Machine Translation*, Nanjing, China, Sep 2009. CWMT. URL http://www.icip.org.cn/cwmt2009/downloads/papers/6.pdf.

Tong Xiao, Qiang Li, Qi Lu, Hao Zhang, Haibo Ding, Shujie Yao, Xiaoming Xu, Xiaoxu Fei, Jingbo Zhu, Feiliang Ren, and Huizhen Wang. The niutrans machine translation system for ntcir-9 patentmt. In *Proceedings of the NTCIR-9 Workshop Meeting*, pages 593–599, Tokyo, Japan, Dec 2011a. NTCIR. URL http://research.nii.ac.jp/ntcir/workshop/OnlineProceedings9/NTCIR/04-NTCIR9-PATENTMT-XiaoT.pdf.

Tong Xiao, Hao Zhang, Qiang Li, Qi Lu, Jingbo Zhu, Feiliang Ren, and Huizhen Wang. The niutrans machine translation system for cwmt2011. In *Proceedings of the 6th China Workshop on Machine Translation*, Xiamen, China, August 2011b. CWMT.

Deyi Xiong, Qun Liu, and Shouxun Lin. Maximum entropy based phrase reordering model for statistical machine translation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 521–528, Sydney, Australia, July 2006. Association for Computational Linguistics. doi: 10.3115/1220175.1220241. URL http://www.aclweb.org/anthology/P06-1066.

Hao Zhang, Liang Huang, Daniel Gildea, and Kevin Knight. Synchronous binarization for machine translation. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 256–263, New York City, USA, June 2006. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N/N06/N06-1033`.

Andreas Zollmann and Ashish Venugopal. Syntax augmented machine translation via chart parsing. In *Proceedings on the Workshop on Statistical Machine Translation*, pages 138–141, New York City, June 2006. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W06/W06-3119`.