Efficient Language Modeling Algorithms with Applications to Statistical Machine Translation

Kenneth Heafield

CMU-LTI-13-017

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213
www.lti.cs.cmu.edu

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
In Language and Information Technologies

Abstract

N-gram language models are an essential component in statistical natural language processing systems for tasks such as machine translation, speech recognition, and optical character recognition. They are also responsible for much of the computational costs. This thesis contributes efficient algorithms for three language modeling problems: estimating probabilities from corpora, representing a model in memory, and searching for high-scoring output when log language model probability is part of the score.

Most existing language modeling toolkits operate in RAM, effectively limiting model size. This work contributes disk-based streaming algorithms that use a configurable amount of RAM to estimate Kneser-Ney language models 7.13 times as fast as the popular SRILM toolkit. Scaling to 126 billion tokens led to first-place performance in the 2013 Workshop on Machine Translation for all three language pairs where submissions were made.

Query speed is critical because a machine translation system makes millions of queries to translate one sentence. Thus, language models are typically queried in RAM, where size is a concern. This work contributes two near-lossless data structures for efficient storage and querying. The first, based on linear probing hash tables, responds to queries 2.42 times as fast as the SRILM toolkit while using 57% of the memory. The second, based on sorted arrays, is faster than all baselines and uses less memory than all lossless baselines.

Searching for high-scoring output is difficult because log language model probabilities do not sum when strings are concatenated. This thesis contributes a series of optimizations that culminate in a new approximate search algorithm. The algorithm applies to search spaces expressed as lattices and, more generally, hypergraphs that arise in many natural language tasks. Experiments with syntactic machine translation show that the new algorithm attains various levels of accuracy 3.25 to 10.01 times as fast as the popular cube pruning algorithm with SRILM.

Acknowledgments

First, I would like to thank my adviser, Alon Lavie, who is primarily responsible for introducing me to machine translation. He especially deserves thanks for supporting a side project on querying language models; that project eventually grew in scope to become this thesis.

Philipp Koehn hosted me at the University of Edinburgh for more than a year and never turned down a travel request. It is because of him that I was able to focus for the final year of this work. He is also one of the organizers of MT Marathon, where some of the chapters started.

It is rare to have a committee that follows—and uses—the work as it is being performed. Chris Dyer provided detailed feedback, encouragement, and help with cdec. He has also been an early adopter and unwitting tester. Bhiksha Raj provided insight into language modeling in general and for speech recognition, including and beyond his own work on the subject.

My colleagues at Carnegie Mellon contributed knowledge and discussions. Jonathan Clark was especially helpful with estimating language models and came up with the name lmplz. Greg Hanneman deserves special mention for contributing his French–English system for use in experiments and answering questions about it. Michael Denkowski has been helpful both with METEOR and evaluations. Victor Chahuneau contributed ideas and code. Thanks to Nathan Schneider, Narges Razavian, Kevin Gimpel, Yubin Kim and the tea group for making it entertaining, even though I rarely drink tea. Kami Vaniea has been especially supportive.

The University of Edinburgh became my adoptive research family. Hieu Hoang motivated me to improve my work and release it to a broader audience. In addition to helping with Moses, he is to credit for the name KenLM. Barry Haddow has been a great resource on the inner workings of Moses. Hieu Hoang, Barry Haddow, Hervé Saint-Amand, and Ulrich Germann were all supportive in research and as officemates. Nadir Durrani's collaboration was essential to the submissions to the Workshop on Machine Translation. Miles Osborne provided his preprocessed copy of the ClueWeb09 corpus and encouraged me to think big with language modeling. Alexandra Birch, Christian Buck, Liane Guillou, Eva Hasler, and Phil Williams have all been helpful in discussions.

MT Marathon is a great venue to work with other researchers. Hieu Hoang, Tetsuo Kiso and Marcello Federico joined my project on state minimization that eventually grew into a chapter. The next year, Ivan Pouzyrevsky and Mohammed Mediani joined my project on estimating language models. Ivan has been particularly helpful with optimizing disk-based algorithms.

System administators from Carnegie Mellon (especially Michael Stroucken), the Texas Advanced Computing Center (TACC) at The University of Texas at Austin, the San Diego Supercomputer Center, and the Unversity of Edinburgh (especially Hervé Saint-Amand and Nicholas Moir) all deserve thanks for handling my requests, such as increasing the maximum size of RAM disks to 700 GB on the machines with 1 TB RAM.

Finally, my users are a source of motivation, ideas, and code. Matt Post, Juri Ganitkevitch, Stephan Peitz, and Jörn Wübker, Jonathan Graehl, and Baskaran Sankaran have all been helpful in discussions and in increasing the impact of this work.

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575, specifically Stampede and Trestles under allocation TG-CCR110017. The author acknowledges the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this thesis. This work made use of the resources provided by the Edinburgh Compute and Data Facility (`http://www.ecdf.ed.ac.uk/`). The ECDF is partially supported by the eDIKT initiative (`http://www.edikt.org.uk/`). The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreements 287658 (EU BRIDGE), 287576 (CASMACAT), 287688 (MateCat), and 288769 (ACCEPT). This material is based upon work supported in part by the National Science Foundation Graduate Research Fellowship under Grant No. 0750271, by the National Science Foundation under grant IIS-0713402, by a NPRP grant (NPRP 09-1140-1-177) from the Qatar National Research Fund (a member of the Qatar Foundation), and by the DARPA GALE program.

Contents

Chapter 1

Introduction

The difference between the right word and the almost right word is the difference between lightning and the lightning bug. –Mark Twain

Practicioners and researchers routinely compromise the quality of their machine translation systems in order to make them tractable. For example, the web has made vast amounts of natural language text available in electronic format and companies such as Google and Microsoft have millions of machines, but their machine translation services use only part of the data and rely on extensive approximations (Brants et al., 2007; Lewis and Eetemadi, 2013). Lewis and Eetemadi (2013) from Microsoft Research summarized the problem: "we now find ourselves the victims of our own success, in that it has become increasingly difficult to train on such large sets of data, due to limitations in memory, processing power, and ultimately, speed". Similarly, researchers routinely mention computational costs as the reason they were unable to make full use of available data, avoid an approximation, or perform an experiment (Bojar et al., 2013b). While computational power is expected to increase over time, researchers have historically applied this power to improve quality by developing more accurate, but often more expensive, models (Chiang, 2007; Schwenk et al., 2012). Efficiency is also critical to practical applications such as speech-to-speech translation on a phone when a traveler does not have Internet access (FEAT Limited, 2012). The primary goal of this thesis is to make statistical machine translation more efficient. The algorithms developed in this work are sufficiently general that they have also been applied to speech recognition (Kim et al., 2012; Si et al., 2013) and optical character recognition (Numen, 2012).

At a high level, statistical machine translation systems have two parts: a translation model that estimates how well the original meaning is preserved and a language model that estimates how fluent the output is (Koehn, 2010). Both models contribute to the computational costs of building systems from data and applying them to translate. This thesis focuses on optimizing the language model. Language models are widely applied in natural language processing, have far more training data than translation models have (because translation models require translated text), and are responsible for most of time cost when models are applied to translate. Chapter 2 explains these points in more detail.

There are many formalisms for language modeling (Jelinek, 1997). This work deals specifically with the successful and widely-used N-gram formalism (Shannon, 1951). While some newer formalisms based on neural networks (Mikolov et al., 2010; Schwenk et al., 2012) have had better results, their practical use is still dependent on N-gram language models. In the N-gram formalism, sentences are conceptually generated one word at a time in natural order (left to right in English). The probability of each word depends on the preceding N–1 words, or a special word for the beginning of sentence. The constant N is known as the order of the model; typical values range from 3 to 6. The probability of a sentence is simply the product of word-level probabilities. While these models are sucessful, computational challenges arise in estimating these models from text and in applying them to translation, where their role is to favor fluent translations.

N-gram language models are estimated from text. Using more text generally leads to improved translation quality, but also increases computational cost. The 2013 Workshop on Machine Translation (Bojar et al., 2013a) provided participants with 7 billion tokens of English text, of which 1 billion tokens had translations in French. Many of the participants did not use all of this data due to computational costs, despite reports from the preceding year that using more data leads to substantial improvements in translation quality (Koehn and Haddow, 2012). Google has shown further translation quality improvements with a language model trained on 1.8 trillion tokens of text but, in order to do so, they used 1500 machines for a day and invented a simpler form of language modeling dubbed stupid backoff (Brants et al., 2007). Part of this thesis, described in Chapter 3, is a more efficient way to estimate state-of-the-art Kneser-Ney language models (Kneser and Ney, 1995; Chen and Goodman, 1998) from large amounts of text. For example, estimating a model on 126 billion tokens of text (Callan et al., 2009) took 2.8 days on one machine. Using this model for translation led to better performance than all other systems, including Google, in the 2013 Workshop on Machine Translation (Bojar et al., 2013a), which bases rankings on human judgments of translation quality.

Statistical machine translation, speech recognition, and optical character recognition systems all build sentences piecemeal from smaller strings. Much of the language model's role is to measure how well these strings flow together (Mohit et al., 2010). Therefore, systems frequently consult the language model during their search for fluent and correct output. For example, a machine translation system queries the language model millions of times per sentence translated. Chapter 4 shows how to make these queries faster while using less RAM than many other methods (Stolcke, 2002), thereby speeding translation and enabling the use of larger, more detailed, language models. The implementation, dubbed KenLM, is being adopted by the translation community as shown in Table 1.1.

Algorithms drive the search for fluent translations. Search can be performed exactly (Bar-Hillel et al., 1964; Iglesias et al., 2011; Rush and Collins, 2011; Aziz et al., 2013), but such algorithms are currently intractable at the scale of modern machine translation systems (Bojar et al., 2013a). For example, Aziz et al. (2013) reported that it took an average of seven hours to translate a seven-word sentence; they did not attempt to translate longer sentences. Approximate search algorithms (Lowerre, 1976; Chiang, 2005; Chiang, 2007; Huang and Chiang, 2007) are therefore in common use; a primary objective of this work is to search more accurately than existing approximate algorithms while consuming less CPU time.

Because the language model examines at most N–1 words of history, it will treat partial sentences the same if they all end with the same N–1 words. Search algorithms exploit this behavior by packing such strings together and efficiently reasoning over all of them at once (Bar-Hillel et al., 1964). Li and Khudanpur (2008) have identified cases where the language model examines even fewer words, permitting the search algorithm to handle even more strings with the same language model queries. This increases the performance of the search algorithm. Chapter 5 contributes an efficient implementation of their work and extends the mechanism to make every query faster.

Approximate search algorithms make pruning decisions based on the relative scores of strings. These algorithms perform better when the scores are more representative of how the string will perform in a sentence. Chapter 6 shows that the common-practice score of a string (Chiang, 2007) is based on an incorrect assumption and shows how to fix the problem, though at the expense of additional memory. Doing so improves search accuracy. Conversely, Chapter 6 also shows to save memory, though string-level scores become less representative and search accuracy decreases. In either case, one can attain the same level of search accuracy by spending less CPU time with improved scores or more CPU time with degraded scores. Therefore, both contributions can be interpreted as time-memory trade-offs.

Strings that begin and end with the same words tend to be treated similarly by the language model when they are extended into full sentences. Chapter 7 contributes a new approximate search algorithm based on this observation. Strings are grouped by common prefixes and suffixes, as shown in Figure 1.1. The algorithm iteratively refines and expands groups that perform well.

Participant	**This Work**	**SRI**	**IRST**	**Berkeley**	**Other**	**Unknown**
BALAGUR (Borisov et al., 2013)			✓			
CMU, CMU-TREE-TO-TREE (Ammar et al., 2013)	✓	✓				
CU-BOJAR, CU-DEPFIX, CU-TAMCHYNA (Bojar et al., 2013c)	✓	✓				
CU-KAREL, CU-ZEMAN (Bílek and Zeman, 2013)		✓				
CU-PHRASEFIX, CU-TECTOMT (Galuščáková et al., 2013)		✓			✓	
DCU (Rubino et al., 2013)	✓	✓	✓			
DCU-FDA (Biciçi, 2013)	✓	✓				
DCU-OKITA (Okita et al., 2013)		✓			✓	
DESRT (Miceli Barone and Attardi, 2013; Miceli Barone, 2013)	✓		✓			
ITS-LATL						✓
JHU (Post et al., 2013)	✓	✓		✓		
KIT (Cho et al., 2013; Niehues, 2013)	✓	✓				
LIA (Huet et al., 2013)		✓				
LIMSI (Allauzen et al., 2013; Allauzen, 2013)		✓			✓	
MES-* (Durrani et al., 2013b; Weller et al., 2013)	✓	✓				
OMNIFLUENT (Matusov and Leusch, 2013)	✓		✓			
PROMT						✓
QCRI-MES (Sajjad et al., 2013; Durrani, 2013)	✓	✓				
QUAERO (Peitz et al., 2013a)		✓				
RWTH (Peitz et al., 2013b)		✓				
SHEF						✓
STANFORD (Green et al., 2013)	✓					
TALP-UPC (Formiga et al., 2012; Formiga et al., 2013)		✓				
TUBITAK (Durgar El-Kahlout and Mermer, 2013)	✓	✓				
UCAM (Pino et al., 2013)	✓	✓			✓	
UEDIN, UEDIN-HEAFIELD (Durrani et al., 2013c)	✓	✓				
UEDIN-SYNTAX (Nadejde et al., 2013)	✓	✓				
UMD (Eidelman et al., 2013)	✓	✓				
UU (Stymne et al., 2013)	✓	✓				
Total	**17**	**22**	**4**	**1**	**4**	**3**

Table 1.1: Adoption of language model toolkits including this work, SRILM (Stolcke, 2002), IRSTLM (Federico et al., 2008), and BerkeleyLM (Pauls and Klein, 2011) by participants in the translation task of the 2013 Workshop on Machine Translation (Bojar et al., 2013b). Participant names and groupings are reproduced from Bojar et al. (2013a). Some participants did not describe their submissions.

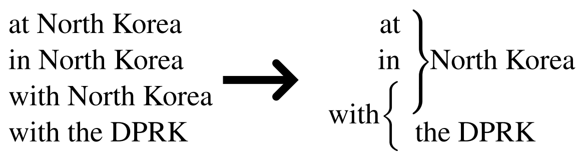


Figure 1.1: Chapter 7 (Search) groups strings by common prefixes and suffixes.

1.1 Research Contributions

This work makes several contributions applied language modeling and search algorithms:

Disk-based Estimation of Kneser-Ney Language Models Chapter 3 develops a series of streaming algorithms that estimate Kneser-Ney (Kneser and Ney, 1995; Chen and Goodman, 1998) language models from text. Because these algorithms are streaming, they make effective use of disk and allow the user to specify how much RAM to use for buffers. The algorithms use orders of magnitude less hardware than previous approaches (Brants et al., 2007), enabling very language language models to be estimated on a single machine. Applying one of these models to machine translation led to first-place performance in a competitive evaluation (Bojar et al., 2013a).

Data Structures for Language Model Queries Chapter 4 contributes two efficient data structures to store and query language models. Both data structures have been designed to reduce memory accesses, employing custom hash tables or interpolation search (Perl et al., 1978) through sorted arrays. Interpolation search formalizes the notion that one opens a dictionary near the end when asked to look up "zebra." The first data structure is designed for speed, but still uses less RAM than the popular SRILM toolkit (Stolcke, 2002). The second data structure saves memory by using sorted arrays and bit-level packing, making it smaller than other lossless options.

Computing Independence Assumptions Search algorithms can efficiently reason over multiple strings at once when they know that the language model will treat the strings the same going forward. Li and Khudanpur (2008) put forward several cases where the language model is guaranteed to treat strings the same. Chapter 5 extends their work by generalizing the criterion and by efficiently computing the criterion. Computation is efficient because the necessary information is precomputed are stored in the sign bit of log probability and log backoff. Finally, Chapter 5 shows how to carry information from one query to the next, making each language model query faster.

Better Scores for Kneser-Ney Models Search algorithms are guided by the scores of strings. Chapter 6 explains why the common-practice score (Chiang, 2007) makes incorrect assumptions and contributes a solution. The solution is to build several language models, each with a different order N, and efficiently store their probabilities alongside the main model, though this costs additional memory. Experiments show that the improved scores lead to improved search accuracy.

Collapsing Probability and Backoff Many language models (Katz, 1987), including Kneser-Ney, store a probability and a backoff with n-grams. Chapter 6 proves that the two values can be collapsed into a single value without changing sentence-level probabilities. This enables language models to be stored in less RAM, though search algorithms are negatively impacted, so more CPU time is required for some tasks.

Faster Search Through Hypergraphs and Lattices Chapter 7 contributes a new approximate search algorithm designed to improve upon cube pruning (Chiang, 2007). Cube pruning makes no effort to exploit the fact that many of the strings it generates share some, but not all, words. The new algorithm groups strings by common prefixes and suffixes. The highest-scoring groups are iteratively refined in a best-first manner. Experiments show a substantial speed improvement compared with cube pruning.

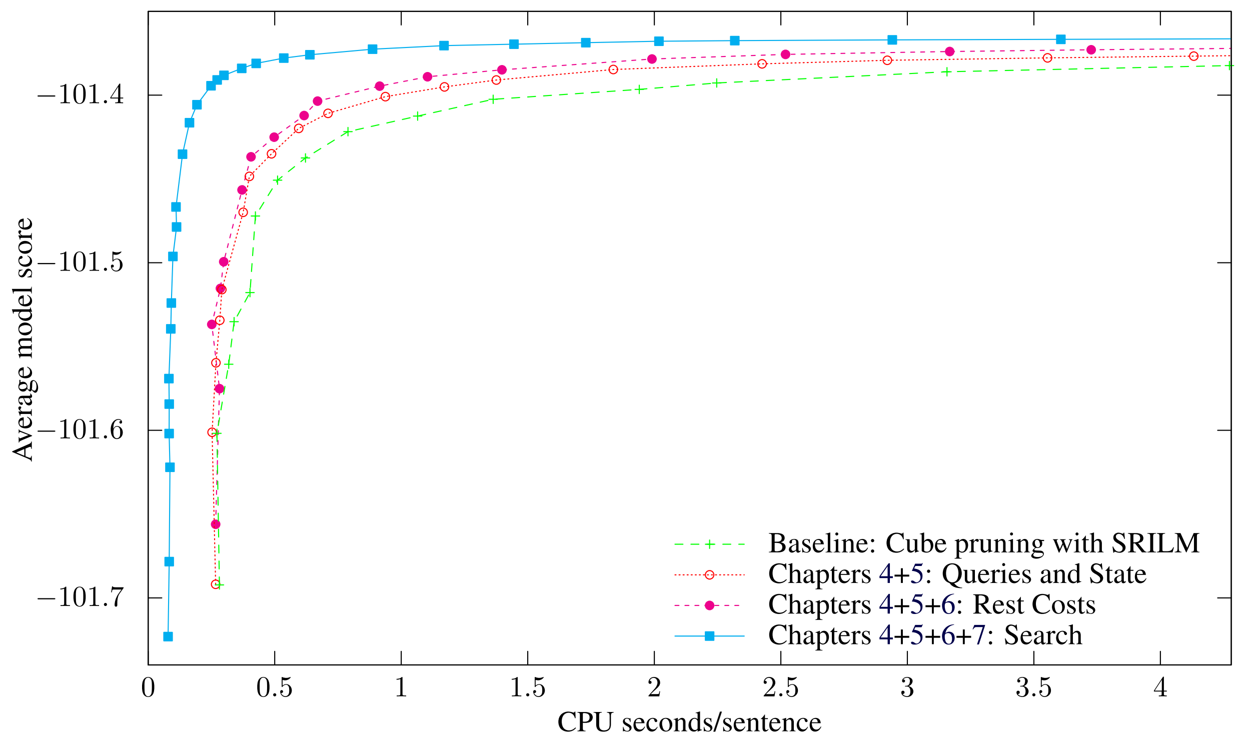
1.2 Preview of Translation Results

Chapters 4, 5, 6, and 7 can be stacked to improve translation performance. Results for a German–English machine translation system (Koehn, 2011), designed for partipation in the 2011 Workshop in Machine Translation (Callison-Burch et al., 2011), are shown in Figure 1.2.

The baseline is common practice prior to this work: the SRI language modeling toolkit (Stolcke, 2002) to query the language model and the popular cube pruning algorithm (Chiang, 2007) for approximate search. Both cube pruning and the new search algorithm in Chapter 7 have a parameter, the beam size, that trades between CPU time and search accuracy. The baseline was run with beam sizes ranging from 5 to 1000, resulting in the bottom curve seen in the figure. The improved options were run with beam sizes starting at 5 and extending until they consumed more time than the baseline at 1000.

The goal of approximate search is to find translations that score highly. Search accuracy can be measured by how well this objective is optimized, namely the score of the translations that it does find. This is known as the model score. Both CPU time and model score are averaged over 3003 sentences. Measurements were made using the popular Moses (Hoang et al., 2009) software; this work has also been implemented in the cdec (Dyer et al., 2010) package. Section 5.6.1 describes the system and methodology in detail.

There is no agreed-upon level of accuracy; a high-frequency trading firm might want speed at the expense of accuracy while researchers generally prefer high accuracy. One way to summarize the results is to ask how much CPU time is required to attain the same level of accuracy. For every baseline data point in Figure 1.2, one can obtain a better model score **3.26 to 10.01 times as fast** by applying the work in this thesis. The speed ratios generally improve at higher levels of accuracy. In this experiment, parameters were optimized for time. Nonetheless, total memory consumption was reduced by 15.3% to 15.8%.



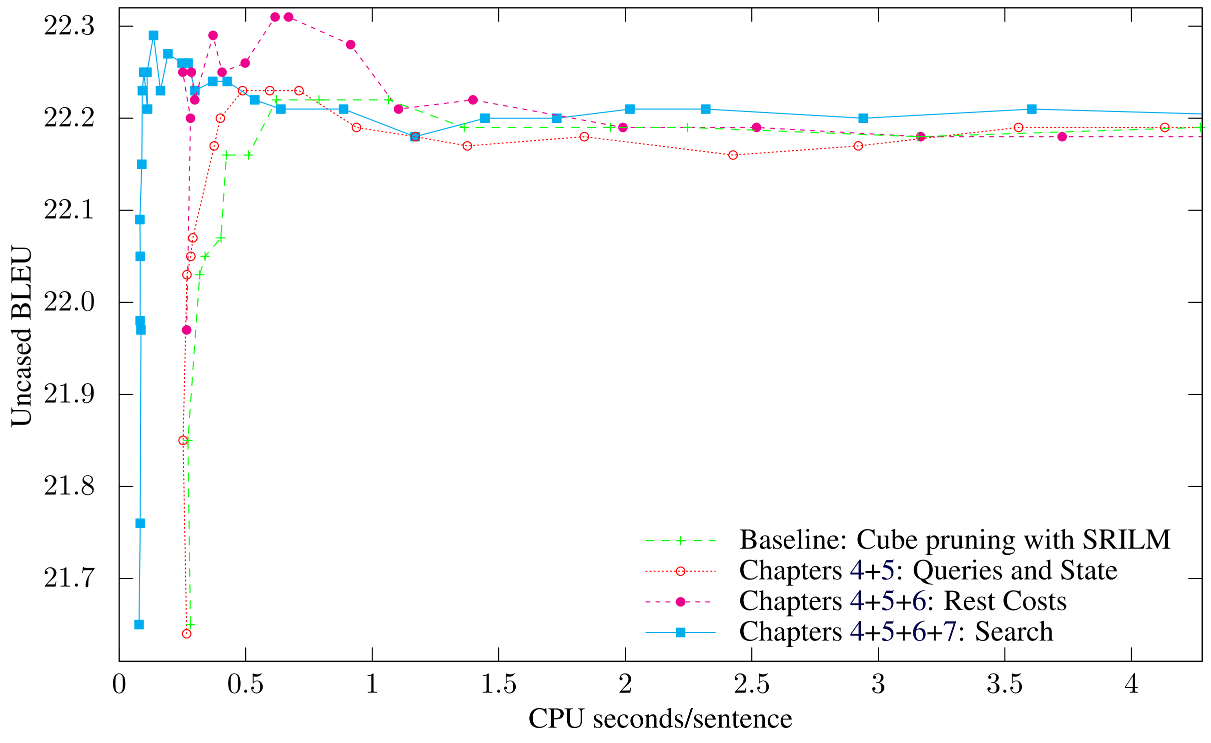


Figure 1.2: Impact of various chapters of this thesis on performance of the Moses (Hoang et al., 2009) software for a German–English translation task (Koehn, 2011). A variety of beam sizes are shown; for the baseline these range from 5 to 1000. Curves near the upper left corner are better. Chapters 4 and 5 are not shown separately because current versions of Moses always use both together; comparisons using older versions appear in the respective chapters.

Chapter 2

Background on Language Models and Decoding

What I tell you three times is true. –Lewis Carroll

When in doubt, tell the truth. –Mark Twain

Language models (Shannon, 1951) estimate how probable text is. This notion is useful to select more fluent translations, likely utterances from a speaker, better spelling suggestions, etc. Log language model probability is therefore often used as a feature in statistical systems, along with other features that model fidelity to the input. Language model probability can also be used as a similarity metric between corpora (Ponte and Croft, 1998; Chong and Specia, 2012).

Given text $w_1\, w_2\, \ldots\, w_{|w|}$ (or $w_1^{|w|}$ for short), the language model probability $p(w_1^{|w|})$ losslessly expands by the chain rule

$$\begin{aligned} p(w_1^{|w|}) = & \, p(w_1) \cdot \\ & p(w_2 \mid w_1) \cdot \\ & p(w_3 \mid w_1\, w_2) \cdot \\ & p(w_4 \mid w_1\, w_2\, w_3) \cdot \\ & p(w_5 \mid w_1\, w_2\, w_3\, w_4) \cdot \\ & p(w_6 \mid w_1\, w_2\, w_3\, w_4\, w_5) \cdot \\ & \vdots \\ & p\left(w_{|w|} \mid w_1^{|w|-1}\right) \end{aligned} \tag{2.1}$$

This expansion is in natural left-to-right order, though text can also be reversed to accomplish right-to-left order (Xiong et al., 2011). The canonical measure of language model performance is *perplexity* (Jelinek et al., 1977). Perplexity is the geometric average probability over some corpus, inverted:

$$\text{Perplexity} = p\left(w_1^{|w|}\right)^{-\frac{1}{|w|}}$$

This is closely related to empirical cross-entropy

$$\log_2 \text{Perplexity} = H\left(w_1^{|w|}, p\right) = \sum_{i=1}^{|w|} \log_2 \frac{1}{p\left(w_i \mid w_1^{i-1}\right)}$$

Lower perplexity means that the language model is better, on average, at predicting each word.

There are many approaches to language models (Jelinek, 1997; Rosenfeld, 2000) i.e. decision trees (Breiman et al., 1984; Bahl et al., 1989), context-free grammars (Moore et al., 1995; Post, 2010; Schwartz, 2012), recurrent neural networks (Mikolov et al., 2010), and N-grams (Shannon, 1951). Decision tree models have largely been abandoned due to disappointing results compared to N-gram models (Rosenfeld, 2000). Using context-free grammars alongside N-gram models results in significantly lower perplexity, but this did not lead to significant improvements in phrase-based machine translation quality (Schwartz, 2012). Recurrent neural networks have shown promising improvements but, due to the high cost of computing probabilities and dependence on full sentence history, have thus far been limited to reranking system outputs or lattices. The candidates for reranking are chosen by using N-gram models (Mikolov et al., 2010). This work is primarily concerned with the dominant N-gram formalism.

2.1 N-gram Models

N-gram models make the Markov independence assumption: only the last $N-1$ words of context matter

$$p\left(w_i \mid w_1\, w_2\, \ldots\, w_{i-1}\right) = p\left(w_i \mid w_{i-N+1}\, w_{i-N+2}\, \ldots\, w_{i-1}\right)$$

If N=5, as it often is, then Equation 2.1 simplifies as

$$\begin{aligned} p(w_1^{|w|}) = p\left(w_1\right) \cdot \\ & p\left(w_2 \mid w_1\right) \cdot \\ & p\left(w_3 \mid w_1\, w_2\right) \cdot \\ & p\left(w_4 \mid w_1\, w_2\, w_3\right) \cdot \\ & p\left(w_5 \mid w_1\, w_2\, w_3\, w_4\right) \cdot \\ & p\left(w_6 \mid w_2\, w_3\, w_4\, w_5\right) \cdot \\ & \qquad\qquad \vdots \\ & p\left(w_{|w|} \mid w_{|w|-4}\, w_{|w|-3}\, w_{|w|-2}\, w_{|w|-1}\right) \end{aligned}$$

The Markov assumption misses phenomena that span arbitrary lengths such as number agreement. Nonetheless, it performs relatively well in terms of perplexity (Rosenfeld, 2000; Chen and Goodman, 1998). In addition to the Markov assumption, it is commonly assumed that sentences are independent. At the beginning of each sentence, context is replaced with the symbol <s>. The end of sentence is explicitly modeled by predicting the special token </s>. Therefore, the probability of sentence $w_1^{|w|}$ is evaluated as

$$p\left(w_1^{|w|} \text{ </s> } \mid \text{ <s>}\right)$$

The token <s> is never predicted; it only appears as context. When systems explicitly generate <s>, the language model interprets it to mean that text should be conditioned on the beginning of sentence.

2.2 Smoothing

Statistical language models may be queried with any combination of words, only some of which appear in their training data. *Smoothing* addresses this problem by reserving some probability for events not seen in the training data. Many smoothing methods employ the backoff algorithm (Katz, 1987). The backoff algorithm assigns probability to word w_n in context w_1^{n-1} according to the recursive equation

$$p(w_n \mid w_1^{n-1}) = \begin{cases} p(w_n \mid w_1^{n-1}) & \text{if } w_1^n \text{ was seen in the training data} \\ b(w_1^{n-1})p(w_n \mid w_2^n) & \text{otherwise} \end{cases} \tag{2.2}$$

where every seen n-gram (up to length N) has a probability p and a backoff b. If an entry is not present or has length exactly N, then its backoff b is implicitly one. Recursion terminates with the special unknown word token <unk>, which has an explicit probability in the model. An example model is shown in Figure 2.1 and an example probability computation is shown in Figure 2.2.

Unigrams

Words	**log *p***	**log *b***
<s>	-∞	-2.0
iran	-4.1	-0.8
is	-2.5	-1.4
one	-3.3	-0.9
of	-2.5	-1.1
<unk>	-6.2	0.0

Bigrams

Words	**log *p***	**log *b***
<s> iran	-3.3	-1.2
iran is	-1.7	-0.4
is one	-2.0	-0.9
one of	-1.4	-0.6

Trigrams

Words	**log *p***
<s> iran is	-1.1
iran is one	-2.0
is one of	-0.3

Figure 2.1: Entries in an example trigram language model with backoff smoothing. For exposition, only selected words are shown.

$\log b$(iran is)	-0.4
$\log b$(is)	-1.4
$\log p$(of)	+ -2.5
$\log p$ (of | iran is)	= -4.3

Figure 2.2: Derivation of $\log p\,(\text{of} \mid \text{iran is})$ according to Equation 2.2 for the model shown in Figure 2.1.

The values of probability p and backoff b are set by the specific smoothing method, which is also responsible for insuring that p is a probability in the sense that it sums to one. There are many smoothing methods in the backoff family i.e. Witten-Bell (Witten and Bell, 1991), Good-Turing (Good, 1953), Jelinek-Mercer (Jelinek and Mercer, 1980), and Kneser-Ney (Kneser and Ney, 1995). Chapter 3 (Estimation) presents an efficient algorithm to estimate probability p and backoff b using interpolated modified Kneser-Ney smoothing (Chen and Goodman, 1998). This is the most popular smoothing technique due to its low perplexity. Huang and Renals (2010) characterized interpolated modified Kneser-Ney as an approximation to a Pitman-Yor process (Pitman and Yor, 1997) and suggested a different modification to Kneser-Ney. Integrating their different modification is proposed as future work.

Some smoothing methods take a different approach to backoff. Stupid Backoff (Brants et al., 2007) is designed to be computed at runtime from counts of n-grams in the training corpus. This makes it easier to estimate than Kneser-Ney because counts are relatively simple to collect. A completely different approach (Schwenk et al., 2012) uses neural networks instead of basing probabilities directly on seen n-grams. Unlike Mikolov et al. (2010), the Markov assumption is still made, albeit with potentially larger N. These continuous space language models are expensive to query and currently only used to rerank system outputs; search is still performed with N-gram models.

The level of assumption made about smoothing differs by chapter. Chapter 3 (Estimation) and the improved rest costs in Chapter 6 (Rest Costs) are specifically designed for Kneser-Ney language models. The remaining work, including the compression technique in Chapter 6 (Rest Costs), is written with backoff models in mind. However, the data structures on Chapter 4 (Queries) could store counts for Stupid Backoff and Chapter 7 (Search) requires only that the model be able to estimate scores given incomplete context.

2.3 Querying

Systems use the language model by querying for conditional probabilities. In order to efficiently compute p according to Equation 2.2, the probability p and backoff b are typically precomputed and stored. When the model backs off, it may have to retrieve up to $N-1$ backoff values in addition to finding the relevant probability. Chapter 5 (State) shows how to avoid retrieving backoff b by carrying information from one query to the next. This optimization applies to applications that make queries left-to-right, such as perplexity computation and often machine translation. Chapter 6 (Rest Costs) presents an alternative that collapses p and b into a single value for each seen n-gram. In all of these cases, there is a need to retrieve values associated with n-grams. Speed is a concern because a machine translation system makes millions of queries per sentence translated, as shown in Section 5.6.4.

By far the fastest and most common way to store language models is lookup tables in local RAM. However, the model must fit into memory alongside the remainder of the application (i.e. a phrase table in machine translation). Chapter 4 (Queries) presents a time- and memory-efficient approach to storing language models in local RAM. There has been a wide variety of work on this problem e.g. SRILM (Stolcke, 2002) and IRSTLM (Federico et al., 2008); baselines are presented in Section 4.3.

Lossy methods use less memory at the expense of accuracy. In some cases, scores returned by the model only approximately sum to one, so perplexity is not well defined. Accuracy can, however, be measured indirectly through end-to-end system performance, such as BLEU (Papineni et al., 2002) in machine translation. Quantization (Whittaker and Raj, 2001) clusters the probability and backoff values then replaces them with their cluster identifiers, thereby saving memory. With 256 clusters, floating-point values take only 8 bits instead of their normal 32 bits and degredation is minimal for speech (Whittaker and Raj, 2001) and machine translation (Federico and Bertoldi, 2006). Optional quantization is included in Chapter 4 (Queries). The compression strategy presented in Chapter 6 (Rest Costs) is lossy in the sense that word-level probabilities are lost but sentence-level scores are unchanged.

Randomization with Bloom filters (Bloom, 1970; Talbot and Osborne, 2007) or minimal perfect hashing (Belazzougui et al., 2008; Talbot and Brants, 2008; Guthrie and Hepple, 2010) substantially reduces memory usage but introduces the risk of false positives. False positives mean that the model thinks it has seen an n-gram and has either created false values or associated it with values from an actual entry. In the event that the value is taken from an actual entry, the impact of the false positive is often minimal because most entries have low probability and the correct probability is usually low as well. Prior work (Talbot and Osborne, 2007) has shown that machine translation is tolerant to some false positives. The present work contributes near-lossless ($\approx 2^{-32}$ false-positive rate due to hash collisions) data structures; extension to lossy data structures is planned as future work.

Pruning, namely removing n-grams, is the most common lossy way to reduce model size. In some cases, careful pruning can even increase performance (Moore and Quirk, 2009). Chelba et al. (2010) studied several ways to prune Kneser-Ney models. Chapter 3 (Estimation) currently does not support pruning, but this is a limitation of the implementation not the algorithm. The remaining work is compatible with pruning. Indeed, Chapter 5 (State) shows how to exploit pruning to improve search.

If the application is tolerant to query latency, then the model can be stored outside local RAM. For example, MITLM (Hsu and Glass, 2008) computes perplexity on a corpus by counting n-grams in the corpus and scanning through the entire model once to retrieve the relevant values. However, many applications decide what queries to make on the basis of results from previous queries. If these queries can be made in batches or work can continue without results for some short time, then the model can be distributed over a network of servers, each holding a subset of n-grams (Brants et al., 2007; Federmann, 2007). Brants et al. (2007) showed how to restructure a machine translation system to batch queries. To maintain throughput, however, each server still needs an efficient local representation in RAM, so much of the work on efficient representation in Chapter 4 (Queries) remains relevant.

2.4 Decoding

In statistical systems, the *decoder* hypothesizes a large set of possible outputs (i.e. translations or transcriptions) and searches for output that scores highly. The score is typically a linear[1] model (Papineni et al., 1997; Och and Ney, 2002) over features including log language model probability.

The following sections describe decoding problems that arise in statistical systems for various tasks. These systems vary substantially in terms of input, search space, and features. However, in terms of search algorithms, they have much in common due to their use of a language model. Section 2.4.2 discusses algorithms for the common search problem. With the exception of Chapter 3 (Estimation), which concerns estimation of language models prior to decoding, the primary goal in this thesis is to improve search in terms of speed, memory, and accuracy.

2.4.1 Decoding Problems

Phrase-Based Machine Translation

In phrase-based machine translation (Koehn et al., 2003), the decoder translates phrases (strings) of input at a time then concatenates them together to form a sentence; an example is shown in Figure 2.3. In general, there are many ways to segment the source sentence into phrases and many ways to translate each phrase into a phrase in the target language. Moreover, the target phrases can be permuted in order to account for differences in word order between languages. In theory, these permutations make the space factorial in the length of the input. In practice, decoders impose a hard limit on the distance that target phrases can move, making the space exponential because there are still multiple ways to translate each source phrase.

Le	*garçon*	*a*	*vu*	*l'*	*homme*	*avec*	*un*	*télescope*
The	boy	seen		man with			the telescope	
The boy		saw		the man		to an telescope		
Some boy		viewed a			men	with a telescope		

Figure 2.3: Fictional phrase-based translations of a French sentence into English. Reordered translations, where the English word order differs from the French word order, are also possible.

Many features make an independence assumption at the boundaries between phrases. In Moses (Koehn et al., 2007), six of the standard features decompose in this way:

1. Log probability of the *target* phrase given the *source* phrase.
2. Log probability of the *source* phrase given the *target* phrase.
3. Word-level log probability of the *target* phrase given words in the *source* phrase.
4. Word-level log probability of the *source* phrase given words in the *target* phrase.
5. Number of phrases used (this is always 1 for an individual phrase pair).
6. Output length (the number of words on the target side of a phrase).

[1]Historically (Papineni et al., 1997), the model was described as log-linear. However, non-probability features like length and event counts make it more natural to think of the model as linear and some of the features as log probabilities. The two descriptions are equivalent.

All of the above features can be expressed as weights on phrase pairs. The feature values sum as phrases are concatenated to form a translation.

Some features cross phrase boundaries. Log probability from a language model biases translations towards fluency and plays an important role in ensuring smooth transitions at phrase boundaries (Mohit et al., 2010). Distortion measures how far a phrase moved when it was reordered. Reordering models (Tillmann, 2004; Galley and Manning, 2008) capture the notion that some phrase pairs, i.e. those that translate French adjectives, move more often than others. An operation sequence model (Durrani et al., 2013a) looks at sequences of word-level changes and movements without regard to phrase boundaries. The operation sequence model borrows much of its infrastructure from language modeling (because both are Markov models) and has recently been optimized by applying Chapter 4 (Queries) (Durrani, 2013).

Experiments with phrase-based machine translation appear in Chapter 3 (Estimation) and Chapter 4 (Queries). Schwartz (2013) has already modified Moses (Koehn et al., 2007) to produce pruned lattices encoding part of the phrase-based search space then applied the search algorithm described in Chapter 7 (Search) to these lattices.

Parsing-Based Machine Translation

Machine translation can be cast, in part, as a parsing problem with a synchronous context-free grammar (Aho and Ullman, 1969; Yamada and Knight, 2001). A synchronous rule consists of one context-free grammar rule for the source language, one context-free grammar rule for the target language, and a bijective correspondence between the non-terminals in these rules. Non-terminals do not need to correspond in order, so entire constituents can be reordered at a time. This explains why parsing-based machine translation performs better than phrase-based machine translation at mid-range reordering (Birch et al., 2009). Example grammar rules are shown in Tables 2.1 and 2.2.

Arabic	**English**
$X \to$ قَالت X_1 ان بيَان في	$X \to$ the X_1 said in a statement that
$X \to X_1$ م■َات X_2	$X \to$ hundreds of X_2 X_1

Table 2.1: Example hierarchical rule pairs from an Arabic-English system (Denkowski, 2013). The grammar uses only X and S, allowing mostly-unconstrained substitution. Subscripts indicate the correspondence between non-terminals.

German	**English**
$X \to X_1$ Europäische Union	$NPB \to DT_1$ European Union
$X \to X_1$ X_2 geschieht ,	$SBAR \to IN_1$ DT_2 is going to happen

Table 2.2: Example target-syntax rule pairs from a German-English translation system (Koehn, 2011) described in Section 5.6.1. The target side has meaningful non-terminal labels.

Sparsity can be a major problem with parsing-based machine translation. Many of the phrases useful for phrase-based machine translation do not correspond to constituents (Koehn et al., 2003), so a grammar rule could not normally be extracted for them. Syntax-augmented machine translation (Zollmann and Venugopal, 2006) addresses this problem by extending the set of grammar labels and parsing algorithm so that phrases can be partial or multiple constituents. Another issue is that, in order to evaluate a non-terminal, both the source and target grammar labels must correspond. Hanneman and Lavie (2013) allows more grammar rules to apply by coarsening the set of labels used. An extreme version of this approach simplifies the set of labels to X and S in one or both languages (Chiang, 2007). Such systems are dubbed *hierarchical* to distinguish them from *syntactic* systems that have several labels for one or both languages. Table 2.1 shows example rules for a hierarchical system where labels on both the source and target side have been simplified.

In principle, the idea is to parse the entire sentence using source rule pairs extracted from training data. xtracted from training data. Due to sparsity, a full parse may be pathological or impossible (i.e. an unknown word). *Glue rules* are a common workaround. These rules allow normal grammar rules to translate pieces of the sentence then glue the pieces together to form a complete translation. The original formulation of glue rules (Chiang, 2007) for hierarchical systems is shown in Table 2.3. The first problem with these rules is that they allow spurious S and $GOAL$ constituents wherever another constituent appears, so decoders typically constrain S to start with the first word of the sentence and $GOAL$ to cover an entire sentence. Doing so enforces that glued hypotheses are built from left to right. The second problem is that intermediate glued hypotheses (those with label S) do not know that they will be preceded by <s>. This makes scores less accurate and prevents Chapter 5 (State) from fully optimizing the hypotheses. Moses (Hoang et al., 2009) developed an improved version of the glue rules, shown in Table 2.4. The improved version natrually enforces left-to-right order (since S can only be built from <s> or another S). Moreover, glued hypotheses contain <s>, so their scores are more accurate and Chapter 5 (State) can optimize them further. Tables 2.3 and 2.4 show rules for hierarchical systems; syntactic systems typically extend the glue rules to accept constituents with labels other than X.

Source	**Target**
$X \rightarrow X_1$	$S \rightarrow X_1$
$X \rightarrow X_1\ X_2$	$S \rightarrow S_1\ X_2$
$GOAL \rightarrow X_1$	$GOAL \rightarrow$ <s> S_1 </s>

Table 2.3: The original glue rules (Chiang, 2007) implemented by default in the cdec decoder (Dyer et al., 2010). To efficiently use these rules, the decoder enforces that S can only cover spans that include the first word of the sentence while $GOAL$ must cover the entire sentence.

Source	**Target**
$X \rightarrow$ <s>	$S \rightarrow$ <s>
$X \rightarrow X_1\ X_2$	$S \rightarrow S_1\ X_2$
$X \rightarrow X_1$ </s>	$S \rightarrow S_1$ </s>

Table 2.4: Improved glue rules originated by the Moses decoder (Hoang et al., 2009) and subsequently adopted by the Joshua (Li et al., 2009) and Jane (Vilar et al., 2010) decoders. Users of cdec (Dyer et al., 2010) can attain better results by overriding the default (Table 2.3) and using these rules, as has been done with all experimental results in this thesis. These rules assume that the input sentence is implicitly surrounded by <s> and </s>.

Parsing with a synchronous context-free grammar is very similar to monolingual parsing (Chappelier et al., 1999), except that non-terminal labels must also match on the target side. Just as with monolingual parsing, the parse can be ambiguous. These ambiguities are encoded in a packed representation, an example of which is shown in Figure 2.4. Formally, this data structure is a directed acyclic hypergraph (Klein and Manning, 2001). A hypergraph is like a graph except that edges can go from one vertex to any number of vertices. In this case, a hyperedge represents a grammar rule application and each vertex is a constituent. Grammar rules can have any number of non-terminals and the hyperedge goes from the constituent it derives (the left hand side of the grammar rule) to all the non-terminals it used (the right hand side of the grammar rule). If a vertex has multiple outgoing edges, that indicates that there are multiple ways to derive a constituent and these have been packed together. This work uses hypergraph terminology consistent with cdec (Dyer et al., 2010), though the parse forest can also be represented as a recursive transition network (Iglesias et al., 2011).

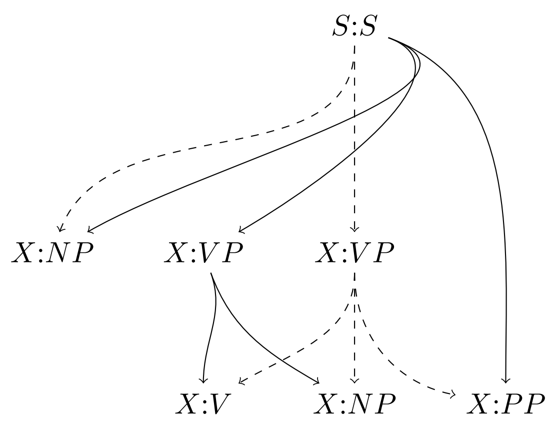


Figure 2.4: An example hypergraph encoding an ambiguous prepositional phrase attachment. The source and target grammar labels are shown separated by a colon. In this work, edges point downward, the same direction as the arrow in context free rules and pointers in the implementation. More phenomena are allowed, including words on edges and permuting constituents during translation.

As described in Chiang (2007), most features are attached to synchronous grammar rules and are analogous to the phrase pair features described in the previous section. Just as in phrase-based machine translation, these feature values sum over the rules used in a derivation. Usually, the language model is the only feature unattached to grammar rules.

It is worth noting that grammar labels and source-language words are not needed for search. All pertinent information has already been encoded in the structure of the hypergraph. Only the structure of the hypergraph, target-language words, and features attached to rules are needed to search for high-scoring translations.

Applications to parsing-based machine translation are the focus of experiments in Chapter 5 (State), Chapter 6 (Rest Costs), and Chapter 7 (Search).

Speech Recognition

Many speech recognition systems perform decoding into two passes: lattice generation and lattice rescoring (Lowerre, 1976). Lattice generation uses a relatively simple language model (typically a trigram model) to predict likely words and score them with an acoustic model that measures agreement with the input. The higher-scoring candidates are encoded in a lattice, an example of which is shown in Figure 2.5. The task in lattice rescoring is to combine the acoustic model with a more detailed language model (often with order N=5) and extract high-scoring hypotheses. However, the language model generates one word at a time, so it systematically assigns lower probability to longer sentences. Adding sentence length as a feature compensates for this effect. The complete scoring model is

$$\text{Score}(d) = \lambda_{\text{Acoustic}} \log \text{Acoustic}(d) + \lambda_{\text{LM}} \log p(\text{Surface}(d)) + \lambda_{\text{Length}} |\text{Surface}(d)|$$

where d is a path through the lattice, Surface(d) is the surface form of the path, and each λ is a feature weight.

Log probability from the acoustic model can be encoded as weights on the edges; any dependence between words was already encoded in the structure of the lattice.

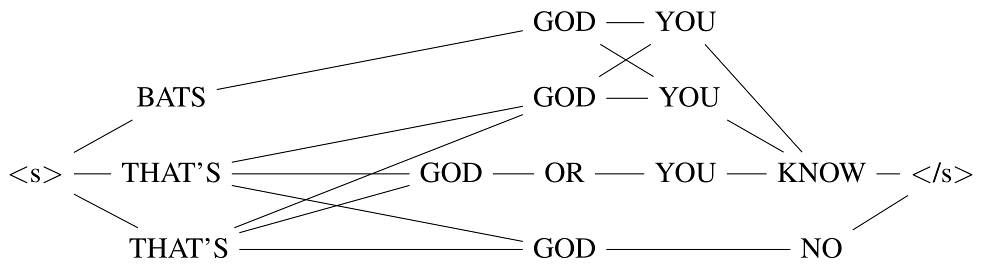


Figure 2.5: An actual speech recognition lattice from the system described in Hasler et al. (2012). Horizontal positions correspond to time but are not to scale; vertical positions are simply for display. The lattice contains many examples of spurious ambiguity. For example, there are two derivations of the output “BATS GOD YOU KNOW”. A small lattice was chosen for illustration; lattices can be much larger.

Chapter 4 (Queries) is already being used in speech recognition (Kim et al., 2012; Si et al., 2013). An adaptation of Chapter 7 (Search) is planned as future work.

Optical Character Recognition

Statistical approaches to optical character recognition (Tong and Evans, 1996; Kolak et al., 2003; Chen et al., 2010) combine character recognition models with language models. Optical features usually examine characters or pairs of characters while a character-level language model biases recognition towards common sequences. When characters are grouped into words, these features simply become weights on the word. However, word segmentation can be ambiguous, a problem Kolak et al. (2003) address jointly by encoding segmentation and recognition options in a weighted finite state transducer. Their representation is equivalent to a lattice (Iglesias et al., 2011).

Optical character recognition literature frequently cites spell checking literature because the two tasks are similar albeit with an optical model instead of a typing model. While unigram language models have been used for spell checking (Kernighan et al., 1990), N-gram language models can flag words that exist but are nonetheless ungrammatical in context (Mays et al., 1991; Church et al., 2007). The same argument explains the use of word-level language models for recognition (Chen et al., 2010).

Numen (2012) has applied the work in Chapter 3 (Estimation), Chapter 4 (Queries), Chapter 5 (State), and Chapter 7 (Search) to optical character recognition lattices.

2.4.2 Searching with Language Models

The search problems described in the preceding sections have much in common. Parsing-based machine translation uses a hypergraph. The other decoding problems can be performed in two passes, where the second pass is searching a lattice. Lattices are simply a special case of hypergraphs where every hyperedge happens to be a normal edge. Thus, the hypergraph search algorithm presented in Chapter 7 (Search) applies to all four problems. However, phrase-based machine translation is typically done in one pass (Koehn et al., 2003; Koehn et al., 2007) without an intervening lattice. Extending Chapter 7 (Search) to directly handle the phrase-based search space is left as future work.

Paths through the search space are called *derivation*s. In phrase-based machine translation, a derivation encodes the way the source sentence was segmented into source phrases, the corresponding target-side phrases chosen for each source phrase, and any permutation made. In parsing-based machine translation, derivations are specific paths through a hypergraph: one way to parse the input and one target-side rule for each source-side rule used in the parse. For lattices, it is a path from beginning of sentence to end of sentence. Derivations have a surface form (from the strings associated with pieces used by the derivation) and a score (a linear combination of feature values). Multiple derivations can have the same surface form, a phenomenon known as spurious ambiguity.

All of the systems described have the same objective: find a highest-scoring surface form (or a list of K top-scoring options) regardless of the ways in which it can be derived. Formally, the goal is to find an output $w_1^{|w|*}$ with highest score

$$w_1^{|w|*} = \arg\max_{w_1^{|w|}} \sum_{d:\text{Surface}(d)=w_1^{|w|}} \text{Score}(d \mid \text{Input})$$

where each d is a derivation. This objective corresponds to a loss function with value zero for perfect output (ignoring the real possibility of multiple perfect outputs) and value one for any other output (Duda and Hart, 1973). Optimizing for other loss functions have been shown to improve quality (Kumar and Byrne, 2004) but, for tractability reasons, such decoding is implemented approximately by reranking a list of K high-scoring hypotheses.

Exact phrase-based decoding with a language model is NP-hard (Sima'an, 1996). Part of the problem is that there are many ways to produce the same translation (i.e. by translating "Le garçon" one word at a time or as a two-word phrase in Figure 2.3) and summing over all these ways is expensive. The common approach is to apply the Viterbi (1967) approximation: search for the highest-scoring derivation and hope that it produces a good translation.

$$w_1^{|w|*} \approx \text{Surface}(\arg\max_d \text{Score}(d \mid \text{Input}))$$

This too is an NP-hard problem (Knight, 1999; Zaslavskiy et al., 2009) but easier in practice to approximately solve. For lattices and hypergraphs, both problems are polynomial (Bar-Hillel et al., 1964) but nonetheless expensive. While there has been some work on avoiding the Viterbi approximation (Demuynck et al., 2002) in speech recognition with trigram language models, most lattice and hypergraph decoders use the Viterbi approximation for tractability.

Searching for high-scoring derivations is difficult due to the language model. Most features are *additive*: when pieces of the search space are assembled, their feature values sum. Were all features additive, search would be relatively simple since pruning decisions could be made exactly and locally. The difficulty arises because *log language model probability is not additive over concatenation*. Non-additivity is the formal algebraic way of saying that e.g. "saw" and "the man" each have their own log language model probability, but these do not sum to form the log probability of "saw the man"

$$\log p(\text{saw}) + \log p(\text{the man}) \neq \log p(\text{saw the man})$$

The reason is that N-gram language models condition on context. In this case, more context in the form of "saw" has been revealed for both "the" and "man". In an example language model,

$$\textsc{Correct}(\text{saw} \bullet \text{the man}) = \frac{p(\text{saw the man})}{p(\text{saw})p(\text{the man})} = \frac{p\,(\text{the} \mid \text{saw})}{p(\text{the})} \frac{p\,(\text{man} \mid \text{saw the})}{p\,(\text{man} \mid \text{the})} = 121.294 \neq 1$$

This equation introduces correction factor Correct, the change in probability when two strings are concatenated. The two strings are delimited by •.[2] Non-multiplicative behavior in the above equation corresponds to non-additive behavior in log space. Chapter 6 (Rest Costs) presents a way to smooth out some

[2]Arguments to a function are traditionally delimited by a comma, which could be mistaken for part of the text.

of this behavior by changing word-level scores without changing sentence-level probabilities. Complete additivity is impossible for general N-gram language models because such a model would be insensitive to word order.

State

When two strings are concatenated, the language model examines at most N–1 trailing words of the first string and at most N–1 leading words of the second string. This is due to the Markov assumption. For example, if the model has order N=4 then

$$\textsc{Correct}(\text{<s> the}\underset{\text{Right State}}{\boxed{\vdash \text{boy saw the}}}\bullet\underset{\text{Left State}}{\boxed{\text{man with a} \dashv}}\text{telescope .}) = \textsc{Correct}(\underset{\text{Right State}}{\boxed{\vdash \text{boy saw the}}}\bullet\underset{\text{Left State}}{\boxed{\text{man with a} \dashv}})$$

where the turnstiles $\vdash$ and $\dashv$ explicitly indicate that no further words will impact the language model probability going forward. In order to facilitate extension on the left or right, a string has both *left state* and *right state* that encode leading and trailing words, respectively. Collectively, left and right state are known as *state*. In some cases, it can be shown that the language model will always back off and fewer words are necessary (Li and Khudanpur, 2008), known as *state minimization*. Chapter 5 (State) shows how to efficiently implement state minimization. Other non-additive features, such as the reordering feature in phrase-based machine translation, maintain their own separate state. In many cases, though, the language model is the only non-additive feature.

If two partial derivations have the same state, then the feature functions will treat them the same going forward. Dynamic programming (Bar-Hillel et al., 1964) exploits this property by *recombining* partial derivations with the same state into a *hypothesis*. Recombination can happen multiple times as derivations are built one edge at a time. If a K-best list of derivations is requested, the a hypothesis remembers at most K partial derivations. State minimization increases recombination opportunities, resulting in improved performance as seen in Chapter 5 (State).

Algorithms

Recombination leads to a straightforward search algorithm (Bar-Hillel et al., 1964) shown in Algorithm 1. Search proceeds by visiting each vertex in topological order (bottom-up in Figure 2.4; left-to-right or right-to-left in Figure 2.5) so that every vertex is visited before those below it. Within a vertex, the algorithm considers every possible way to apply the outgoing edges to form hypotheses. Hypotheses with identical state are recombined. The algorithm proceeds until it has reached the final vertex, where it can unpack the top-scoring hypotheses to yield a K-best list. The problem with this algorithm is that there are $O(|\text{vocabulary}|^{2N-2})$ possible language model states to consider at each vertex. This term is quite large for typical vocabulary sizes (100 thousand to 393 million) and orders N (3 to 6), making the algorithm intractable for large search problems.

Some search algorithms (Koehn et al., 2007; Watanabe et al., 2006; Huang and Mi, 2010) are designed to build derivations from left to right. In that case, only right state is necessary.[3] This reduces the number of possible states to $|\text{vocabulary}|^{N-1}$, still intractable with the straightforward dynamic programming approach.

Beam search (Lowerre, 1976), shown in Algorithm 2, approximates the straightforward algorithm by pruning to the top k hypotheses after processing each vertex.[4] The parameter k is known as *beam size*. For

[3]This also happens naturally as a result of state minimization. All partial derivations start with the beginning of sentence symbol <s>. No N-gram predicts <s>, so state minimization determines that all words can be omitted from left state.

[4]Throughout this thesis, K refers to the number of complete derivations requested by the user and k refers to the beam size used during search.

```
for each vertex $v$ in topological order do
    for each outgoing hyperedge $e$ from $v$ do
        /* Consider all ways to apply the rule associated with $e$     */
        $e_1^{|e|}$ ← the ordered tuple of vertices that $e$ points to.
        for each combination of hypotheses $(h_1, \ldots, h_{|e|}) \in$ hyps$[e_1] \times \cdots \times$ hyps$[e_{|e|}]$ do
            /* Create hypothesis $y$ by performing rule application.    */
            $y \leftarrow$ApplyRule$(e, h_1^{|e|})$
            /* Add hypothesis $y$ to the set at vertex $v$, recombining if
               an existing hypothesis has the same state.               */
            if another hypothesis $x \in hyps[v]$ has state $s(x) = s(y)$ then
                $x \leftarrow \arg\max_{z \in \{x,y\}}$ Score$(z)$
            else
                hyps$[v] \leftarrow$ hyps$[v] \cup \{y\}$
            end
        end
    end
end
```

Algorithm 1: The straightforward bottom-up dynamic programming algorithm (Bar-Hillel et al., 1964) to find the single-best derivation. Not shown is the subroutine ApplyRule that generates a hypothesis by applying the grammar rule, phrase, or lattice transition associated with a hyperedge.

```
for each vertex $v$ in topological order do
    for each outgoing hyperedge $e$ from $v$ do
        $e_1^{|e|}$ ← the ordered tuple of vertices that $e$ points to.
        for each combination of hypotheses $(h_1, \ldots, h_{|e|}) \in$ hyps$[e_1] \times \cdots \times$ hyps$[e_{|e|}]$ do
            $y \leftarrow$ApplyRule$(e, h_1^{|e|})$
            if another hypothesis $x \in hyps[v]$ has state $s(x) = s(y)$ then
                $x \leftarrow \arg\max_{z \in \{x,y\}}$ Score$(z)$
            else
                hyps$[v] \leftarrow$ hyps$[v] \cup \{y\}$
            end
        end
    end
    Prune hyps$[v]$ to the top $k$ hypotheses by score.
end
```

Algorithm 2: Beam search (Lowerre, 1976) adds a pruning step to the straightforward algorithm, making it approximate.

$k < |\text{vocabulary}|^{2N-2}$, there is no guarantee that an optimal derivation will be found because the language model is non-additive. Smaller k make pruning more aggressive, increasing the risk that some part of an optimal derivation will be pruned. Larger k costs more CPU time (because there are more rule applications) but generally leads to higher-scoring results (because each pruning pass preserves more hypotheses). Thus k trades between time and accuracy.

The approach taken by beam search is somewhat wasteful: it generates many hypotheses then discards all but the top k. Cube pruning (Chiang, 2007) takes a different approach: only generate k hypotheses to begin with. More formally, it pops k hypotheses off the top of a priority queue. Initially, the queue is populated with the best (highest-scoring) hyperedges with the best hypotheses from each vertex referenced by the hyperedges. When a hypothesis is popped, several next-best alternatives are pushed. These alternatives substitute the next-best hyperedge or a next-best hypothesis from one of the vertices. The beam size k is also known as the *pop limit* because it is the number of hypotheses popped off the priority queue for each vertex.

Cube pruning performs better than beam search in terms of the time-accuracy trade-off. Moreover, there are several variations on the theme (Huang and Chiang, 2007; Petrov et al., 2008; Gesmundo and Henderson, 2010) discussed further in Section 7.2. Nonetheless, computational issues continue to prevent practicioners from making full use of data or experimenting with new ideas (Chung and Galley, 2012; Zeman, 2012). Chapter 4 (Queries), Chapter 5 (State), and Chapter 6 (Rest Costs) all present ways to reduce the language model's contributions to the computational costs of search, with experiments in cube pruning. Chapter 7 (Search) presents an alternative to cube pruning with improved time-accuracy performance that also builds on the preceding chapters.

2.4.3 Implementations

Moses (Koehn et al., 2007; Hoang et al., 2009), Jane (Vilar et al., 2010), Johsua (Li et al., 2009), and cdec (Dyer et al., 2010) all implement cube pruning for machine translation. These implementations can be compared by decoding the same input with the same model, taking care to ensure that scores are comparable.[5]

Decoder settings were optimized in consultation with their developers (Hoang, 2013; Peitz et al., 2013c; Post, 2013; Dyer, 2013). Moses was used with the in-memory phrase table. Chapter 4 (Queries) and Chapter 5 (State) are used in all results; for some decoders there is no other option. Only default cube pruning implementations are shown; variations on cube pruning are discussed in Chapter 7 (Search). All decoders were run with beam sizes ranging from 5 to at least 2000. At beam size 2000, Joshua is the slowest, so other decoders were run with larger beam sizes to fill in the plots.

Figures 2.6 and 2.7 show results using the hierarchical German–English system and methodology described in Section 5.6.1. Moses is the primary baseline because it is the fastest.[6] Decoders perform similarly in terms of accuracy with the same beam size. This is not an accident, but rather the product of several rounds of experiments where developers adopted practices from other decoders, such as the improved glue rule (Table 2.4) from Moses and Chapter 5 (State). In terms of speed, Moses, Jane, and Joshua all have similar constant costs but cdec's constant cost is much higher. The difference may be that cdec applies every target-side grammar rule at least once. Other decoders score target-side grammar rules in isolation, sort by score at loading time,[7] and allow cube pruning to determine which rules to apply. With regard to RAM,

[5]Costs in Jane were negated to form scores. All decoders used the improved glue rule from Table 2.4, so their feature definitions are compatible up to feature weighting (and weights were scaled appropriately). Score equivalence was confirmed by checking that short sentences had exactly the same derivation and the same score.

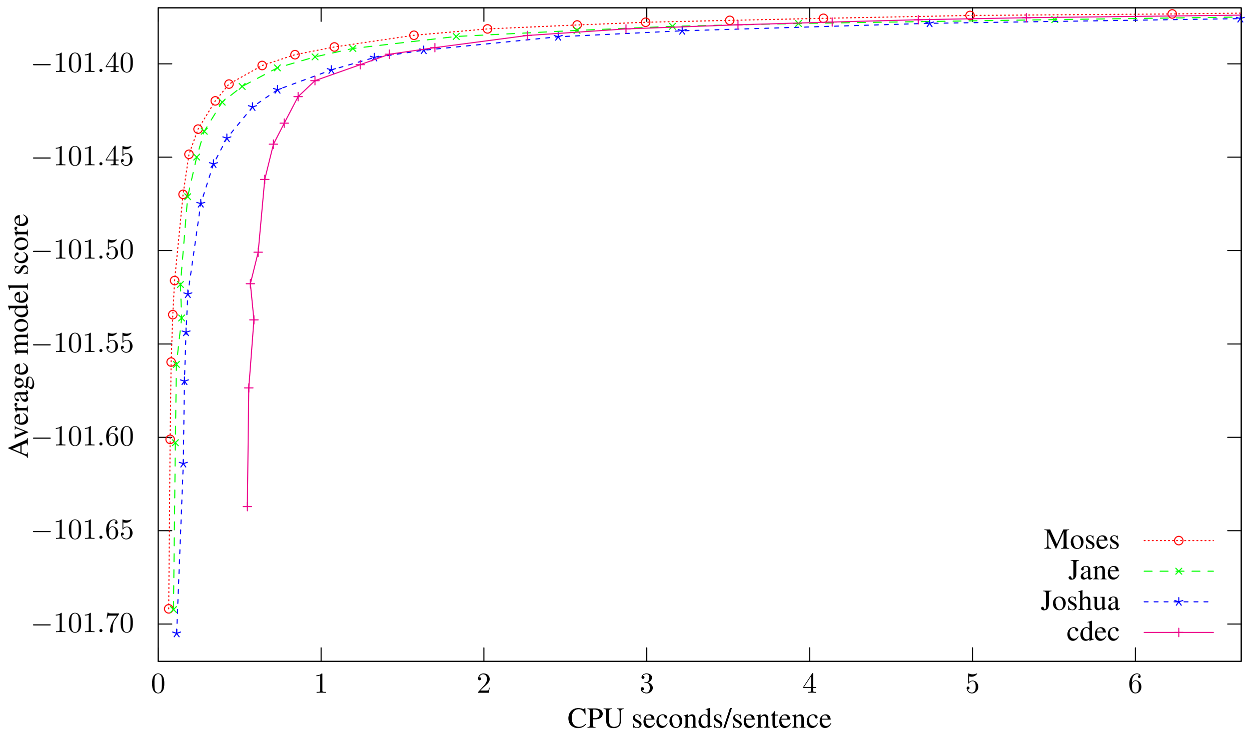
[6]Results presented in this chapter use a recent version of Moses, specifically revision 78cdf8. The remainder of the thesis uses older versions from the time the work was performed. Older versions are slower at parsing, so they have a higher constant cost. Since much of the thesis reports speed ratios and constants bias the ratios towards one, the ratios are even better with newer versions.

[7]Joshua can delay sorting until rules are actually needed. This option was disabled in order to minimize decoding time. Decoding time excludes loading time.

Jane has the most memory-efficient rule table, so its initial memory consumption is the lowest, followed by cdec. However, Moses has the lowest slope, using only 0.14 GB more RAM with beam size 2000 than with beam size 5. The difference is 1.07 GB for Jane and 1.60 GB for cdec. This suggests that Moses represents hypotheses more compactly.

2.5 Summary

Language models are a crucial component of many natural language processing systems. They are also responsible for many of the computational costs associated with training and running these systems. The algorithms presented in this thesis reduce computational costs at every stage in the estimation and application of language models in a statistical system.



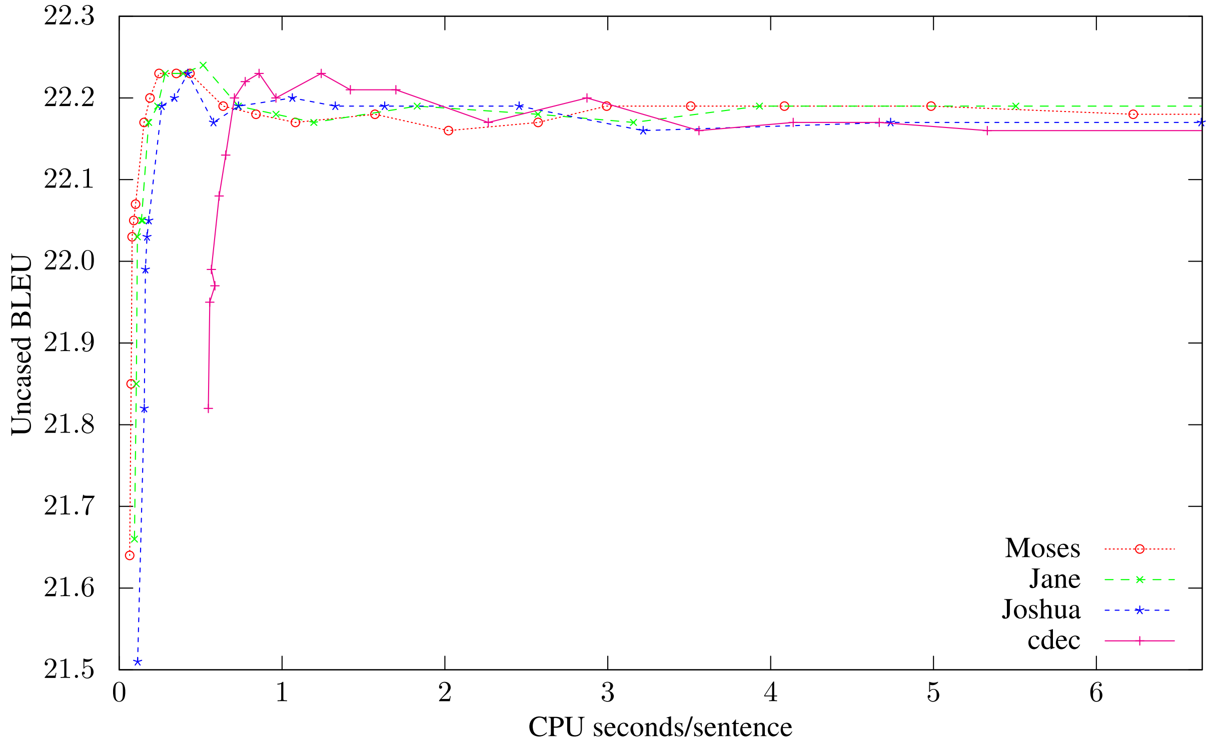


Figure 2.6: Time-accuracy trade-off presented by cube pruning implementations for the hierarchical German–English machine translation system.

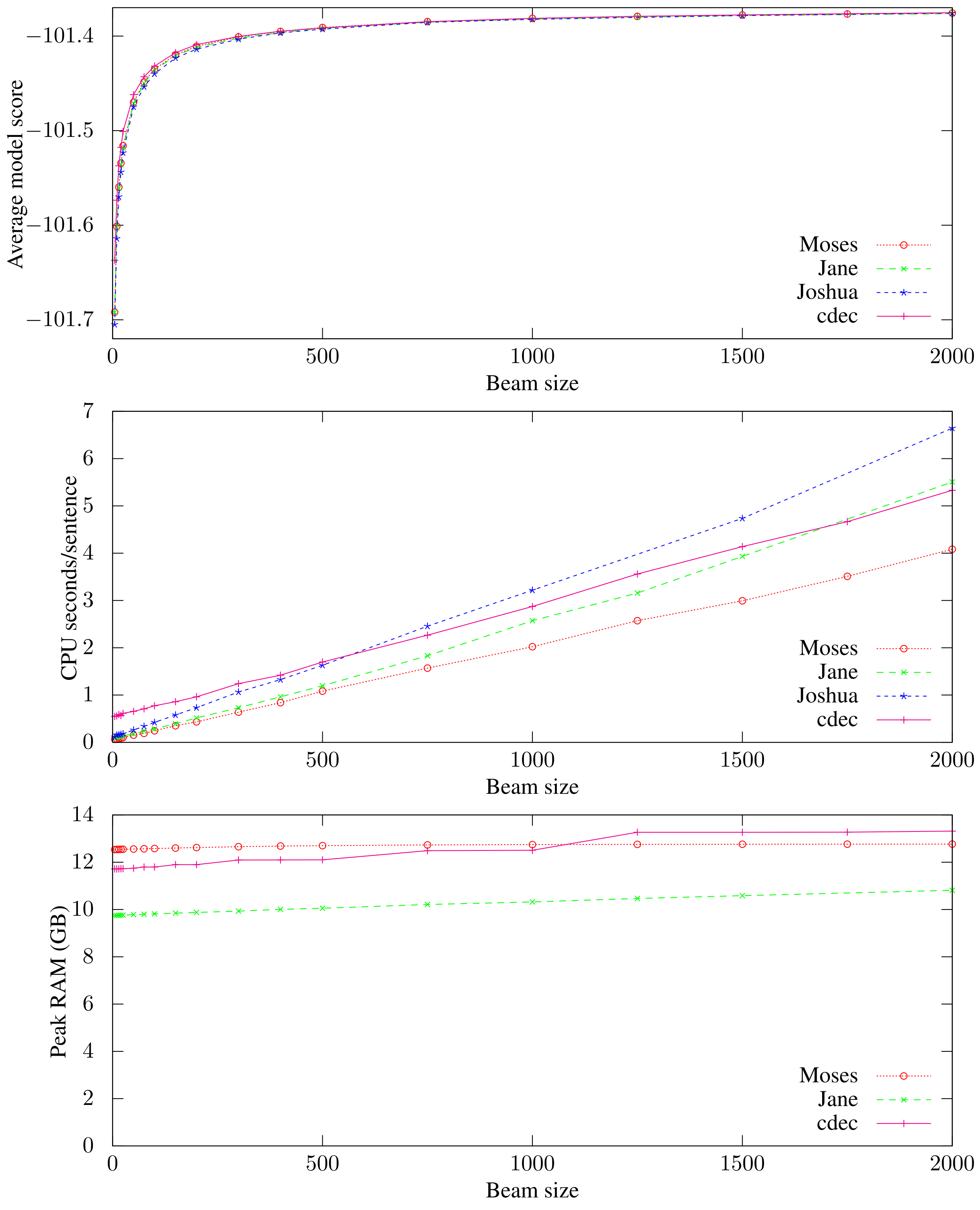


Figure 2.7: Performance of decoder implementations by beam size for the hierarchical German–English machine translation system. Joshua's memory consumption is not shown because it is written in Java. Java requires the user to guess how much memory a process will use, so a fair comparison would require running binary search over this parameter for all beam sizes.

Chapter 3

Estimating Kneser-Ney Language Models[1]

If you laid all of our laws end to end, there would be no end. –Mark Twain

Delay not, Caesar. Read it instantly. –William Shakespeare, *Julius Caesar*

Estimating a large language model with Kneser-Ney smoothing previously required large amounts of RAM (Stolcke, 2002), approximation (Federico et al., 2008), or machines (Brants et al., 2007). As a result, practicioners used less data (Callison-Burch et al., 2012; Biciçi, 2013) or simpler smoothing methods (Brants et al., 2007). This chapter presents a disk-based streaming approach that uses user-configurable RAM, makes no approximation other than limited floating-point rounding, and runs on one machine.[2]

On a corpus with 126 billion tokens, estimation took 2.8 days wall time on one machine with 140 GB RAM. Using this large language model (via Chapter 4) led to first-place performance in the 2013 Workshop on Machine Translation (Bojar et al., 2013a). Moreover, estimation is faster than memory-based options offered by SRILM (Stolcke, 2002) and IRSTLM (Federico et al., 2008), taking 14.0% and 9.0% of the wall time, respectively, on a corpus with 302 million tokens.

3.1 Introduction

As explained in Section 2.2, interpolated modified Kneser-Ney smoothing (Kneser and Ney, 1995; Chen and Goodman, 1998) is popular due to its relatively low perplexity. This chapter shows how to estimate these models efficiently. No effort is made it improve or approximate the smoothing method; the result is exactly the same as in Chen and Goodman (1998) but equations have been rewritten to clearly explain each step. SRILM's (Stolcke, 2002) implementation differs slightly from the literature in whether it interpolates unigrams; both options are explained and implemented.

More formally, given some corpus and an order N, the task in this chapter is to estimate probability p for all n-grams up to length N and backoff b for all n-grams up to length $N - 1$ in the corpus. The smoothing equations define these values as a function of n-gram counts. Typical orders N range from 3 to 6.

The challenge is that there are many n-grams (121 billion in one experiment) and keeping track of them, along with intermediate values, is expensive. Many toolkits (Stolcke, 2002; Federico et al., 2008) use in-memory data structures similar to those used for querying; a discussion of these data structures appears in Section 4.3. At first, this seems acceptable because there is little value to estimating a language model that cannot be queried. However, query data structures can make optimizations that do not easily apply to estimation:

[1]A prior version of this work was published as Heafield et al. (2013b).

[2]An optional cluster version is planned as future work.

- The data structures presented in Chapter 4 assume that the number and set of unique n-grams is known from the start. Neither of these is known while the corpus is being read.
- If system inputs are known in advance, the language model can be filtered to remove n-grams that will not be queried (Heafield and Lavie, 2010).
- Low-count n-grams could be pruned to save space. Calculating Kneser-Ney smoothing statistics requires counts of all singleton n-grams.
- Quantization (Whittaker and Raj, 2001) can save memory by approximating floating-point values. Quantizing intermediate values during estimation would lead to compound errors.

This chapter approaches the problem by storing n-grams on disk. The performance properties of disk, such as the cost of seeking, play a substantial role in algorithm design. Section 3.3 shows how computation is broken into four streaming passes and three sorts. Section 3.5 describes how streaming and sorting are optimized for disk.

3.2 Related Work

3.2.1 Google

Brants et al. (2007) showed how to estimate Kneser-Ney models with a series of five MapReduces (Dean and Ghemawat, 2004). On 31 billion words, estimation took 400 machines for two days. Recently, Google estimated a pruned Kneser-Ney model on 230 billion words (Chelba and Schalkwyk, 2013), though no cost was provided.

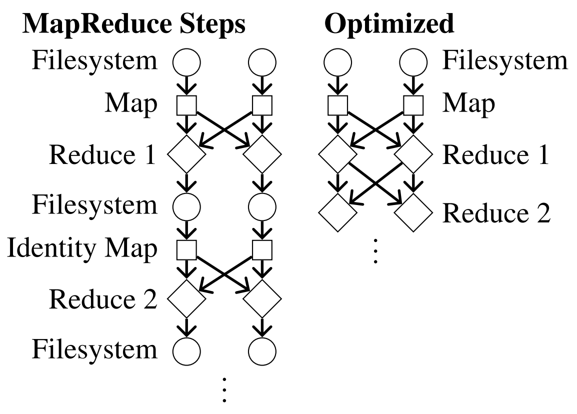


Figure 3.1: Each MapReduce performs three copies over the network when only one is required. Arrows denote copies over the network (i.e. to and from a *distributed* filesystem). Both options use local disk within each reducer for merge sort.

Each MapReduce consists of one layer of mappers and an optional layer of reducers. Mappers read from a network filesystem, perform optional processing, and route data to reducers. Reducers process input and write to a network filesystem. Ideally, reducers would send data directly to another layer of reducers, but this is not supported. Their workaround, a series of MapReduces, performs unnecessary copies over the network (Figure 3.1). In both cases, reducers use local disk.

Writing and reading from the distributed filesystem improves fault tolerance. However, the same level of fault tolerance could be achieved by checkpointing to the network filesystem then only reading in the case of failures. Doing so would enable the next stage of processing to start without waiting for the network filesystem to write all the data.

The present work runs on a single machine using local disk, avoiding many of the engineering issues that a generalized MapReduce implementation would entail. These are left as future work. However, Appuswamy et al. (2013) have noted that single machines can be more cost effective than clusters in terms of both raw performance and performance per dollar.

Brants et al. (2007) contributed Stupid Backoff, a simpler form of smoothing calculated at runtime from counts. With Stupid Backoff, they scaled to 1.8 trillion tokens. Stupid Backoff is cheaper to estimate, but this work makes Kneser-Ney practical. Another advantage of Stupid Backoff has been that it stores one value, a count, per n-gram instead of probability and backoff. Chapter 6 shows how to collapse probability and backoff into a single value without changing sentence-level probabilities.

3.2.2 Microsoft[3]

MSRLM (Nguyen et al., 2007; Gao and Clark, 2013) aims to scalably estimate language models on a single machine. Counting is performed with streaming algorithms similar to this work. Their parallel merge sort also has the potential to be faster. The biggest difference is that their pipeline delays some computation until query time. In particular, the numerator and denominator of normalization (Section 3.3.3) are stored in RAM and divided at query time, costing more memory and CPU time. Interpolation (Section 3.3.4) is also performed at query time. Due to this query-time processing, MSRLM is incompatible with other language model toolkits.

In MSRLM, files are accessed via memory mapping. Entire files at mapped at a time and these files may be larger than physical RAM. Without explicit guidance, the kernel uses heuristics to decide what portions of the files to leave on disk. Computation will inevitably read parts of the file that are not in memory, causing the kernel to block computation until the read operation is complete. Section 3.5 explains how this work uses dedicated threads for reading and writing, explicitly informing the kernel what parts to keep in RAM and performing disk operations in parallel with computation.

3.2.3 SRI

SRILM (Stolcke, 2002) is a popular language modeling toolkit that implements interpolated modified Kneser-Ney among other smoothing methods. Estimation is typically performed by storing n-grams in RAM. Users are given the choice between two memory formats: the default based on hash tables or the slightly more compact format based on sorted arrays.

Counting, adjusting counts, and computing summary statistics (the equivalent of Sections 3.3.1 and 3.3.2) can also be done with a text-based on-disk pipeline. However, subsequent smoothing steps are done in RAM. If pruning is enabled, some pruned n-grams need not be loaded in RAM because the summary statistics have already been computed. Without pruning, both options have the same peak RAM usage and the on-disk pipeline takes longer.

3.2.4 IRST

IRSTLM (Federico et al., 2008) is a free language modeling toolkit. It does not implement interpolated modified Kneser-Ney but rather an approximation dubbed "modified shift-beta".[4] Among other approximations, it skips the adjusted count step (Section 3.3.2) and uses normal counts, reducing estimation time (Bertoldi, 2013).

[3]Unfortunately, MSRLM does not compile and run on recent versions of Linux, so empirical comparisons are omitted.

[4]Earlier versions refer to their approximation as "improved Kneser-Ney", where "improved" means easier to estimate.

By default, IRSTLM processes the entire corpus in RAM. It can also process pieces of the corpus at a time. Steps communicate using plain text files as opposed to the binary formats in this work. When pieces are processed in serial, peak memory usage is below that of normal IRSTLM. The disadvantage is that approximation increases with the number of pieces.

3.2.5 MIT

MITLM (Hsu and Glass, 2008) is specifically designed for language model estimation. Like this work, it implements interpolated modified Kneser-Ney smoothing using closed-form estimates for discounts (Chen and Goodman, 1998). Optionally, it can explicitly optimize the discounts on training data, yielding models with lower perplexity than this work can. Since the benchmarks are concerned solely with performance, the faster closed-form estimate is used in experiments.

Entries are stored in large vectors, one for each order. The data structure is similar to a trie except that the pointers are reversed: each n-gram record w_1^n contains a pointer to the record for its history w_1^{n-1}. This is designed to make it efficient to retrieve statistics about the history of an n-gram during estimation. All data structures are stored in RAM.

3.2.6 Berkeley

BerkeleyLM (Pauls and Klein, 2011) is more recent but uses the most memory, is slower than comparable memory-based offerings from SRILM and MITLM, and does not implement Kneser-Ney smoothing. As described in Section 3.3.2, Kneser-Ney smoothing requires that discounts be estimated from data (Kneser and Ney, 1995). The latest revision (Pauls, 2013) of BerkeleyLM uses constant discounts of 0.75, so the smoothing method might be more accurately described as absolute discounting. Moreover, it does not implement interpolation or modified discounting (Chen and Goodman, 1998). Estimation is performed in RAM.

3.3 Estimation Pipeline

Estimation has four streaming passes: counting, adjusting counts, normalization, and interpolation. Data is sorted between passes, three times in total. Figure 3.2 shows the flow of data.

3.3.1 Counting

In order to estimate a language model of order N, this step counts all N-grams (with length exactly N) by streaming through the corpus. Words near the beginning of sentence also form N-grams padded by the begin of sentence marker <s> (possibly repeated multiple times). The end of sentence marker </s> is appended to each sentence and acts like a normal token.

Unpruned N-gram counts are sufficient for subsequent steps, so lower-order n-grams ($n < N$) are not counted. Conversely, unpruned N-gram counts are necessary to compute smoothing statistics.

Strings are mapped to vocabulary indices with a linear probing hash table (Knuth, 1963).[5] Keys in the hash table are 64-bit hashes (Appleby, 2012) of the surface strings.[6] The strings are not retained in RAM. However, unique strings are written to disk for later use in the final model file.

[5]This hash table is the only part of the pipeline that can grow. Users can specify an estimated vocabulary size for purposes of memory budgeting. If vocabulary size becomes an issue, future work could use separate vocabularies for each block of input then merge them later.

[6]If two strings have the same hash, then they will be assigned the same vocabulary identifier, an instance of the birthday paradox. However, for a vocabulary of size 393 million and a high-quality 64-bit hash function (Appleby, 2012), there is approximately 99.5% chance that no collision will occur.

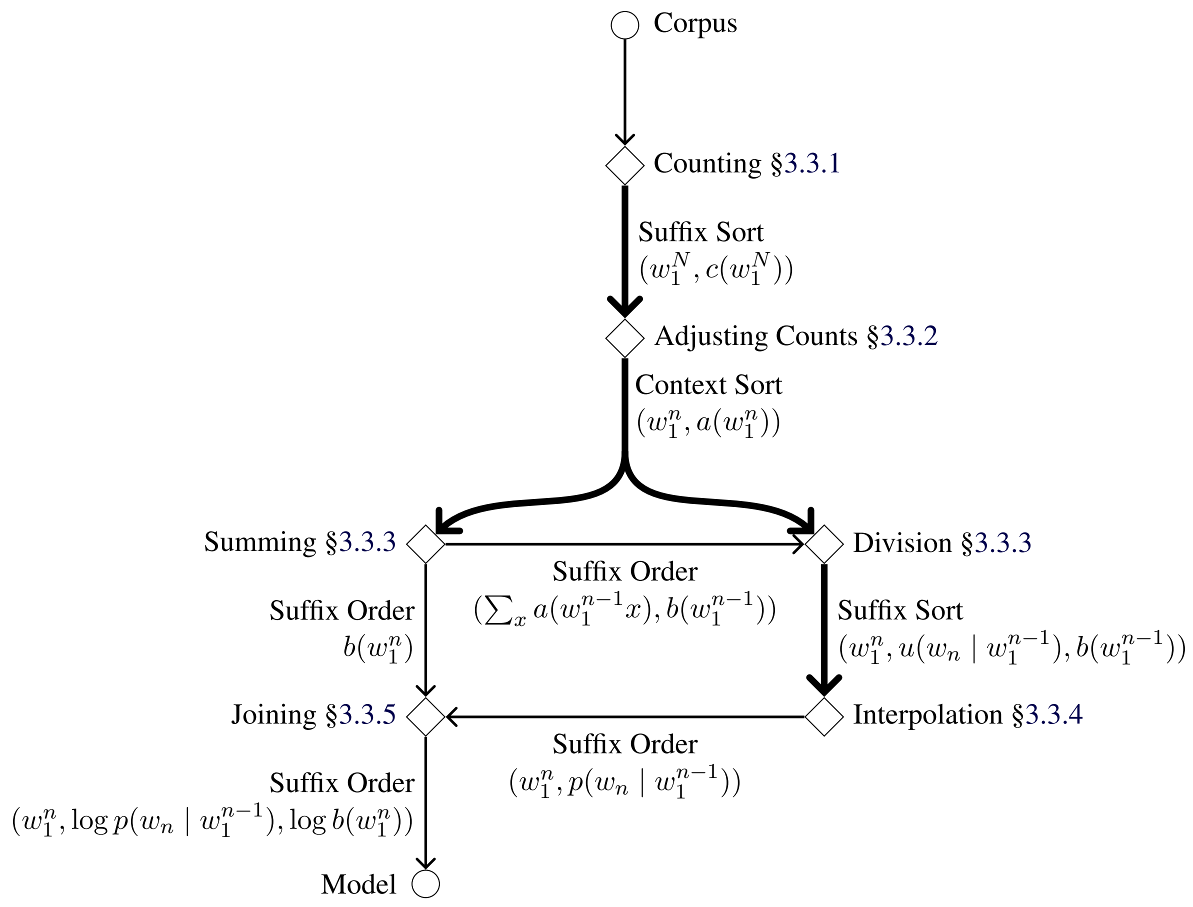


Figure 3.2: Data flow in the estimation pipeline. Normalization has two threads per order: summing and division. Interpolation and joining happen in one thread but are conceputally different steps. Thick arrows indicate sorting. Vertical arrows indicate data is stored on disk. Horizontal arrows indicate data is passed in memory between simultaneous steps.

This step outputs blocks of counted N-grams at a time. An exact hash table is used to identify duplicate N-grams within a block so that they can be combined. Merge sort also combines identical N-grams (Bitton and DeWitt, 1983).

3.3.2 Adjusting Counts

The counts c are replaced with adjusted counts a.

$$a(w_1^n) = \begin{cases} c(w_1^n) & \text{if } n = N \text{ or } w_1 = \text{<s>} \\ |\{v : c(v w_1^n) > 0\}| & \text{otherwise} \end{cases}$$

The two cases arise from the backoff algorithm (Section 2.2). When a language model is queried, it will attempt to match a full-length N-gram. As a special case, n-grams bound to the beginning of sentence may be short simply because there is nothing beyond the beginning of sentence token. The counts of these entries are preserved. In all other cases, the language model will only consult an n-gram because it was unable to find a longer match. Kneser-Ney smoothing therefore conditions the probability of lower-order entries on

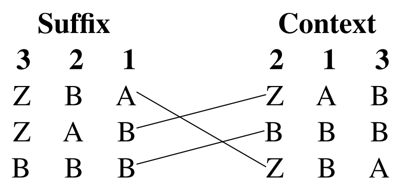


Figure 3.3: In suffix order, the last word is primary. In context order, the penultimate word is primary.

backing off. This is accomplished by adjusting the count. Chapter 6 points out that traditional methods for scoring sentence fragments (Chiang, 2007) ignore this aspect of Kneser-Ney smoothing and proposes a solution.

Adjusted counts are computed by streaming through N-grams sorted in suffix order (Figure 3.3). The algorithm keeps a running total of $a(w_i^N)$ for each i and compares consecutive N-grams to decide which adjusted counts to increment or output. Formally, if the previous and current N-grams are respectively $v_1^n w_{n+1}^N$ and w_1^N with $v_n \neq w_n$, then $a(w_{n+1}^N)$ is incremented and $a(v_1^n w_{n+1}^N), a(v_2^n w_{n+1}^N), \ldots, a(v_n^n w_{n+1}^N)$ are output before being reset to 1. The beginning of sentence token is a special case.

Smoothing statistics are also collected in preparation for the next step. For each length n, it collects the number $t_{n,i}$ of n-grams with adjusted count $i \in [1, 4]$.

$$t_{n,i} = |\{w_1^n : a(w_1^n) = i\}|$$

These are used to compute closed-form estimates (Chen and Goodman, 1998) of discounts $D_n(i)$

$$D_n(i) = \begin{cases} 0 & \text{if } i = 0 \\ i - \dfrac{(i+1)t_{n,1}t_{n,i+1}}{(t_{n,1} + 2t_{n,2})t_{n,i}} & \text{if } 1 \leq i \leq 3 \\ D_n(3) & \text{if } i > 3 \end{cases}$$

Less formally, adjusted counts 0 (the unknown word) through 2 have special discounts.

The discounts must fall in a valid range in order for smoothing to continue.

$$0 \leq D_n(i) \leq i \, \forall i \in [0, 3], n \in [1, N]$$

This is not a theorem but rather an assumption. It may fail on short corpora or those that contain duplicated text. Discounts can also be out of range when Kneser-Ney smoothing is not appropriate, such as part of speech language models where there is no singleton unigram. In these cases, a different smoothing strategy should be used.

Specialized discounts are what distinguish modified Kneser-Ney smoothing from ordinary Kneser-Ney smoothing, where a single discount is used. Huang and Renals (2010) propose a third set of discounts. MITLM (Hsu and Glass, 2008) determines discounts empirically instead of using a closed-form estimate.

3.3.3 Normalization

Normalization computes pseudo probability u

$$u(w_n \mid w_1^{n-1}) = \frac{a(w_1^n) - D_n(a(w_1^n))}{\sum_x a(w_1^{n-1}x)}$$

and backoff b

$$b(w_1^{n-1}) = \begin{cases} \frac{D_n(1)C_n(1) + D_n(2)C_n(2) + D_n(3)C_n(3)}{\sum_x a(w_1^{n-1}x)} & \text{if } w_1^{n-1} \text{ is the context of some } n\text{-gram} \\ 1 & \text{otherwise} \end{cases} \tag{3.1}$$

where the backoff statistics are specialized for low counts

$$C_n(i) = \begin{cases} |\{x : a(w_1^{n-1}x) = i\}| & \text{if } i \leq 2 \\ |\{x : a(w_1^{n-1}x) \geq 3\}| & \text{if } i = 3 \end{cases} \tag{3.2}$$

The difficulty lies in computing denominator $\sum_x a(w_1^{n-1}x)$ for all w_1^{n-1}. To make this computation efficient, the n-grams are sorted in context order (Figure 3.3) so that, for every w_1^{n-1}, the entries $w_1^{n-1}x$ are consecutive. One pass collects both the denominator and backoff statistics $C_n(i)$ for $i \in [1, 3]$.

A problem arises in that denominator $\sum_x a(w_1^{n-1}x)$ is known only after streaming through all $w_1^{n-1}x$, but is needed immediately to compute each $u(w_n \mid w_1^{n-1})$. One option is to buffer in memory, taking $O(N|\text{vocabulary}|)$ space since each order is run independently in parallel. Instead, there are two threads for each order. The sum thread reads ahead to compute $\sum_x a(w_1^{n-1}x)$ and $b(w_1^{n-1})$ then places these in a secondary stream. The divide thread reads the input and the secondary stream then writes records of the form

$$(w_1^n, u(w_n \mid w_1^{n-1}), b(w_1^{n-1})) \tag{3.3}$$

The secondary stream is short so that data read by the sum thread will likely be cached when read by the divide thread. This sort of optimization is not possible with most MapReduce implementations.

The next step, interpolation, needs backoff $b(w_1^{n-1})$ to compute final probability $p(w_n \mid w_1^{n-1})$. However, the final model (Section 3.3.5) places backoff $b(w_1^n)$ next to $p(w_n \mid w_1^{n-1})$. One way to make both available with w_1^n would be to write records of the form

$$(w_1^n, u(w_n \mid w_1^{n-1}), b(w_1^{n-1}), b(w_1^n))$$

These records are bigger, which would cost more to sort. Moreover, one thread processes n-grams $w_1^{n-1}x$ to compute $b(w_1^{n-1})$ while another thread processes $(n+1)$–grams $w_1^n x$ to compute $b(w_1^n)$. These threads would have to be synchronized to merge their output. Instead, the backoffs $b(w_1^n)$ are written to N files, one for each $0 \leq n < N$, as bare values without keys. Because the entries $w_1^n x$ are sorted in context order, their contexts w_1^n are sorted in suffix order. Therefore, these files contain backoffs $b(w_1^n)$ in suffix order, which will be useful in Section 3.3.5.

3.3.4 Interpolation

Chen and Goodman (1998) found that perplexity improves when the various orders within the same model are interpolated. The interpolation step computes final probability p according to the recursive equation

$$p(w_n \mid w_1^{n-1}) = u(w_n \mid w_1^{n-1}) + b(w_1^{n-1})p(w_n \mid w_2^{n-1}) \tag{3.4}$$

Recursion terminates when unigrams are interpolated with the uniform distribution

$$p(w_n) = u(w_n) + b(\epsilon)\frac{1}{|\text{vocabulary}|}$$

where ϵ denotes the empty string. The unknown word counts as part of the vocabulary and has count zero,[7] so its probability is

$$p(\text{<unk>}) = \frac{b(\epsilon)}{|\text{vocabulary}|}$$

Probabilities are computed by streaming in suffix lexicographic order: w_n appears before w_{n-1}^n, which in turn appears before w_{n-2}^n. In this way, $p(w_n)$ is computed before it is needed to compute $p(w_n \mid w_{n-1})$, and so on. This is implemented by jointly iterating through N streams, one for each length of n-gram. The relevant pseudo probability $u(w_n \mid w_1^{n-1})$ and backoff $b(w_1^{n-1})$ appear in the input records (Equation 3.3).

3.3.5 Joining

This step unites probability $p(w_n \mid w_1^{n-1})$ and backoff $b(w_1^n)$ so that they can be stored together in the final language model. Interpolation (Section 3.3.4) computes $p(w_n \mid w_1^{n-1})$ in suffix order by using $b(w_1^{n-1})$, so it does not know $b(w_1^n)$. Normalization (Section 3.3.3) stored backoffs $b(w_1^n)$ in suffix order. Joining simply merges the probability stream from interpolation with the backoff files written by normalization.[8]

Chapter 4 describes query-time data structures that store language models. One of these, the trie data structure, is built by first sorting n-grams in suffix order. Since records are already sorted in this order,[9] the trie data structure can be built in the same pass as interpolation. However, if quantization (Whittaker and Raj, 2001) is enabled, one pass is required to train the quantizer while a second pass applies the quantizer.

3.4 Numerical Precision

Exact integer arithmetic is used where possible. In particular, the smoothing statistics $t_{n,i}$ in Section 3.3.2, the normalization denominator $\sum_x a(w_1^{n-1}x)$ in Section 3.3.3, and backoff terms $C_n(i)$ in Section 3.3.3 are all exact integers. Thus, no precision is lost when performing $O(|\text{vocabulary}|)$ operations to arrive at these statistics. Non-integer values such as the discounts D, pseudo probabilities u, backoffs b, probability p, and particularly $p(\text{<unk>})$ are all computed from exact integers in $O(N)$ floating-point operations.

In contrast, SRILM (Stolcke, 2002) computes backoff b from probabilities[10] according to the equivalent equation

$$b(w_1^n) = \frac{1 - \sum_{x:w_1^n x \text{ appears}} p(x \mid w_1^n)}{1 - \sum_{x:w_1^n x \text{ appears}} p(x \mid w_2^n)}$$

and computes the unknown word probability as

$$p(\text{<unk>}) = 1 - \sum_{w \neq \text{<unk>}} p(w)$$

[7]SRILM implements "another hack" that computes $p_{\text{SRILM}}(w_n) = u(w_n)$ and $p_{\text{SRILM}}(\text{<unk>}) = b(\epsilon)$ whenever $p(\text{<unk>}) < 3 \times 10^{-6}$, as it usually is. Both are implemented in this work. Their motivation may have been numerical precision.

[8]Backoffs will only appear in the file produced by Section 3.3.3 if the n-gram is the context of some $(n+1)$–gram. If an n-gram is not a context (the unknown word or the end of sentence $w_n = \text{</s>}$), then the backoff is zero and the backoff file position is not advanced.

[9]The trie data structure requires that vocabulary identifiers be assigned in a particular way and that sorting order be based on its vocabulary identifiers. Moreover, this assignment requires that all words be known at the time. After the vocabulary is known (namely, counting is completed), a vocabulary identifier mapping is created. This mapping is applied during the adjusted counts step, so no extra pass or sort is required.

[10]SRILM computes backoffs according to Equation 3.1, uses them to compute probabilities in Equation 3.4, forgets the backoffs, and then reconstructs backoffs from probabilities. This may have been done to generically support multiple smoothing methods or pruning.

These calculations are numerically imprecise because $O(|\text{vocabulary}|)$ floating-point operations are performed to arrive at a values that are small compared to rounding error. In some cases, SRILM can generate backoffs $b(w_1^{n-1}) > 1$, but this is impossible when the discounts are in range and the model is not pruned. Nonetheless, such backoffs may be valid for pruned models or those smoothed with other methods, so the data structures in Chapter 4 preserve the ability to represent backoffs greater than one (or zero in log space).

Proposition 1. *If the discounts $D_n(i)$ are in range for all orders n and counts i*

$$0 \le D_n(i) \le i \; \forall i \in [0,3], n \in [1,N]$$

then backoff $b(w_1^{n-1}) \le 1 \; \forall w_1^{n-1}$ in an unpruned language model with interpolated modified Kneser-Ney smoothing.

Proof. Equation 3.2 defined counts-of-adjusted counts $C_n(i)$ for each order n and count $i \le 3$. These counts are non-negative, so both sides of the assumption can be multiplied to yield

$$D_n(i)C_n(i) \le iC_n(i) \; \forall i \in [1,3], n \in [1,N] \tag{3.5}$$

Equation 3.2 has two cases for $C_n(i)$: adjusted counts $i \le 2$ and adjusted counts 3 or above. In the first case where $i \le 2$:

$$\begin{aligned} iC_n(i) &= |\{x : a(w_1^{n-1}x) = i\}| \\ &= i \sum_{x:a(w_1^{n-1}x)=i} 1 \\ &= \sum_{x:a(w_1^{n-1}x)=i} i \\ &= \sum_{x:a(w_1^{n-1}x)=i} a(w_1^{n-1}x) \end{aligned}$$

where $a(w_1^n)$ is the adjusted count of w_1^n. By transitivity with Equation 3.5,

$$D_n(i)C_n(i) \le \sum_{x:a(w_1^{n-1}x)=i} a(w_1^{n-1}x) \; \forall i \in [1,2], n \in [1,N]$$

where D_n are the discounts for order n. The second case, where $i = 3$, is similar:

$$\begin{aligned} 3C_n(3) &= |\{x : a(w_1^{n-1}x) \ge 3\}| \\ &= 3 \sum_{x:a(w_1^{n-1}x)\ge 3} 1 \\ &= \sum_{x:a(w_1^{n-1}x)\ge 3} 3 \\ &\le \sum_{x:a(w_1^{n-1}x)\ge 3} a(w_1^{n-1}x) \end{aligned}$$

where a is adjusted counts. By transitivity with Equation 3.5,

$$D_n(i)C_n(i) \le \sum_{x:a(w_1^{n-1}x)\ge i} a(w_1^{n-1}x) \; \text{ for } i = 3, \forall n \in [1,N]$$

where again D_n are the discounts for order n. Summing cases where the inequality holds,

$$\sum_{i=1}^{3} D_n(i)C_n(i) \leq \left(\sum_{x:a(w_1^{n-1}x)=1} a(w_1^{n-1}x)\right) + \left(\sum_{x:a(w_1^{n-1}x)=2} a(w_1^{n-1}x)\right) + \left(\sum_{x:a(w_1^{n-1}x)\geq 3} a(w_1^{n-1}x)\right)$$
$$= \sum_x a(w_1^{n-1}x) \tag{3.6}$$

It remains to show that $\sum_x a(w_1^{n-1}x) > 0$ by cases.

1. w_1^{n-1} is not a context of any n-gram. Then backoff $b(w_1^{n-1}) = 1$ by definition (Equation 3.1) and the claim holds.
2. w_1^{n-1} is a context but adjusted count $a(w_1^{n-1}x) = 0 \forall x$. The only entry with adjusted count zero is the unknown word. Thus $w_1^{n-1}x = $ <unk>$\forall x$. This implies that the model consists solely of the unknown word and the training data is empty. Kneser-Ney smoothing is not defined in this case.
3. w_1^{n-1} is a context and adjusted count $a(w_1^{n-1}x) > 0$ for some x. The other terms are non-negative, so the sum is positive.

Dividing both sides of Equation 3.6 by a positive number,

$$\frac{\sum_{i=1}^{3} D_n(i)C_n(i)}{\sum_x a(w_1^{n-1}x)} \leq 1$$

Recognizing the definition of backoff from Equation 3.1,

$$b(w_1^{n-1}) \leq 1$$

□

3.5 Streaming and Sorting Framework

This section describes a general streaming and sorting framework along with how it is applied to support language model estimation. It is designed to use handle fixed-length records, support simultaneous streams, use user-specified memory, and optimize sorting.

For language model estimation, records encode n-grams. An n-gram record is an array of n vocabulary identifiers (4 bytes each, as the largest vocabulary to date is 393 million words) and an 8-byte value. The value can be a count or two floating-point numbers. The various orders $1 \leq n \leq N$ are handled as separate streams, so records within a stream have fixed length, eliminating the need to store length information. However, record length is determined at runtime, which is not supported by two existing frameworks (Vengroff, 1994; Dementiev et al., 2008).[11]

Processing steps are threads. Records are passed between threads using producer-consumer queues. To reduce overhead, records are grouped into blocks and these blocks are passed by reference. Processing steps can choose whether to process blocks (i.e. to combine counts of n-grams within a block) or simply perceive a stream of records. When the last step is finished with a block, it is recycled.

Records are updated in-place, improving upon MapReduce implementations that typically copy. The current pipeline does not need to change the size of records (an 8-byte count conveniently takes the same

[11]These frameworks require that records have a type that does not use memory allocation. Technically, it is possible to create a type for each order n up to some fixed limit with C++ templates, but doing so makes the pipeline steps unnecessarily complicated.

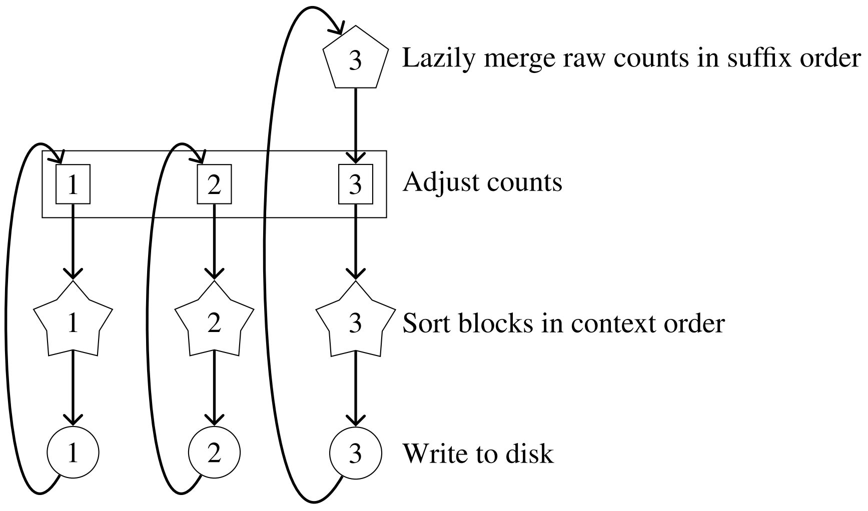

3
Lazily merge raw counts in suffix order
1
2
3
Adjust counts
1
2
3
Sort blocks in context order
1
2
3
Write to disk

Figure 3.4: The adjusted counts step in detail for a trigram language model (N=3). There are N circular chains of processing steps, one for each order. Input N-gram counts are lazily merged in suffix order for consumption by the adjusted counts algorithm. A single thread computes adjusted counts by manipulating N streams. In preparation for the next step, normalization, blocks are sorted in context order before they are written to disk. Blocks are then recycled. There are $2 + 2N$ threads in total: one lazy merge sort, one adjusted counts algorithm, N block sorters, and N disk writers.

space as two floating-point values) but, if it did, this could be accomplished by reading input from one stream and writing to another.

Much work has been done on efficient disk-based merge sort. Particularly important is arity, the number of blocks that are merged at once. Low arity leads to more passes while high arity incurs more disk seeks. Abello and Vitter (1999) modeled these costs and derived an optimal strategy: use fixed-size read buffers (one for each block being merged) and set arity to the number of buffers that fit in RAM, leaving enough space for a write buffer. The optimal buffer size is hardware-dependent; 64 MB is the default. To overcome the operating system limit on file handles, multiple blocks are stored in the same file.

Merge sort is further optimized by using the streaming framework as shown in Figure 3.4. If sufficient memory is available for read buffers, the last merge is performed lazily in memory. Output blocks are sorted in the next step's desired order before they are written to disk. These optimizations eliminate up to two copies to disk if enough RAM is available. Moreover, disk operations happen in lazy merge sort and disk writing threads, which execute in parallel with computation.

Because lazy merge sort is not easily amenable to overwriting its input, peak disk usage is twice the amount required to store all n-gram records. Additional costs are null-delimited vocabulary strings and the backoff files written in Section 3.3.3.

Due to the use of merge sort, complexity is $O(|n\text{-grams}| \log |n\text{-grams}|)$. In theory, an algorithm based on hash tables would have linear complexity $O(|n\text{-grams}|)$. However, on-disk hash tables have a much higher constant cost due to seeking.

3.6 Experiments

3.6.1 Methodology

Experiments use the ClueWeb09 (Callan et al., 2009) corpus collected from websites. After spam filtering (Cormack et al., 2011), removing markup, selecting English, splitting sentences (Koehn, 2005), deduplicating, tokenizing (Koehn et al., 2007), and truecasing, 126 billion tokens remained.

Benchmarks were run on an otherwise-quiet machine using local disk. Peak virtual memory usage is measured by the kernel and collected immediately before termination. CPU time is the sum of system and user time reported by the kernel. Toolkits were configured to write their own binary files, except for IRSTLM's disk-based pipeline and BerkeleyLM where it is more efficient to write ARPA files.

3.6.2 Memory Setting

With this work, the user determines how much memory to use. Figure 3.5 shows how this setting impacts CPU and wall time costs. The task is to estimate a 5-gram language model on sentences randomly sampled, without replacement, from ClueWeb09. The test machine has 64 GB RAM and 32 cores. The memory setting impacts both block sizes and the arity of merge sort as explained in Section 3.5. In turn, these determine the number of merge sort passes and whether there is enough RAM to do the last pass lazily. With 1.2 GB RAM, the sorting framework falls back doing a full merge, writing to disk, and later reading the merged output. Wall time increases because more data is written to disk. It also increases because merge sort and streaming are performed serially rather than in parallel.

3.6.3 Toolkit Comparison

Toolkit performance was measured by building an unpruned 5–gram language model. Results from the previous section are comparable because the same machine and subsets of ClueWeb09 were used; for simplicitly, only the 3.9 GB setting is shown in comparisons.

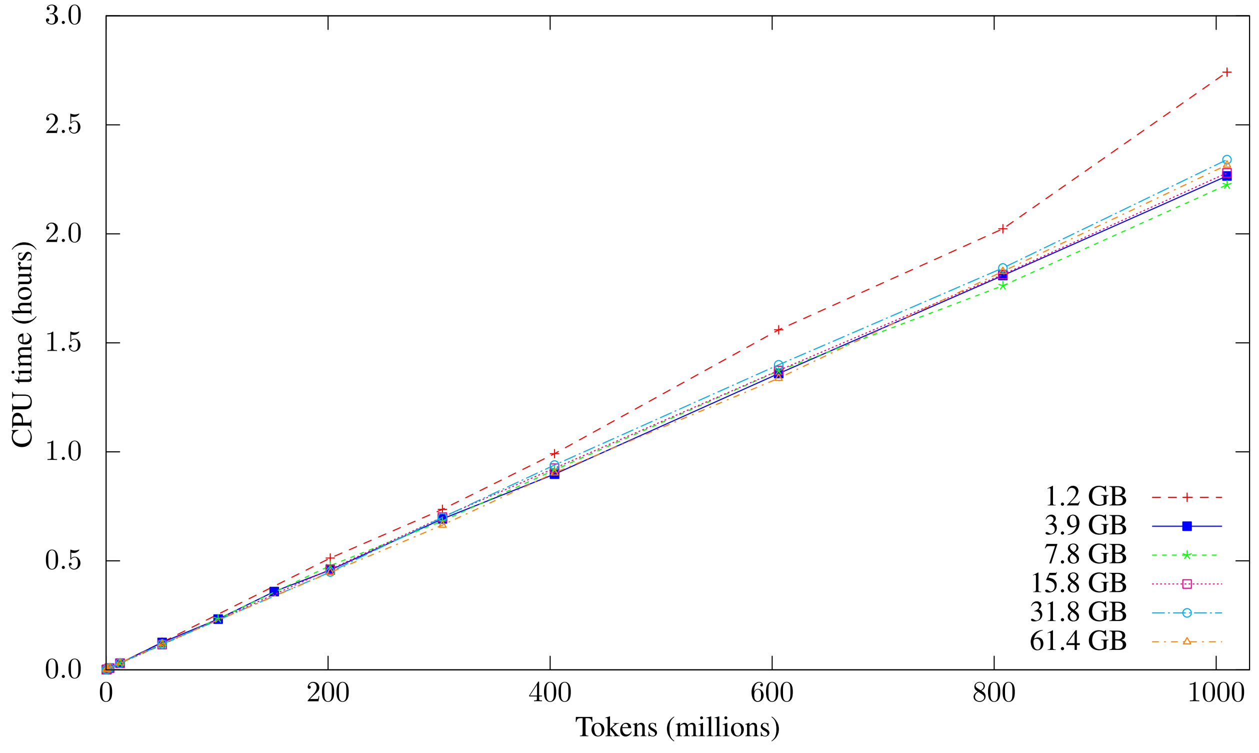
While most toolkits run in a single process, SRILM and IRSTLM also offer pipelines that entail a series of processes. In these cases, the memory usage reported is the maximum of any one process (as if they were run serially). The CPU time is summed and actual wall time is reported (allowing for any parallelism). Thus, the combination of memory usage and wall time may not be feasible, but represents a lower bound on resource usage. IRSTLM's disk pipeline has a parameter, the number of pieces, that trades between approximation and peak memory (were steps run in serial). Three pieces were used in the experiments.

With BerkeleyLM, one is required to guess, in advance, how much memory it will use. Guessing too low leads to a crash (in contrast, this work performs more merge passes). Guessing too high would inflate the statistics reported. Binary search was used to find the lowest setting, in integer gigabytes, with which it would run. The process's peak virtual memory consumption is reported as usual. To confirm that this setting did not negatively impact CPU time, the experiment was also run with plentiful memory.

Results are in Figure 3.6. Compared with SRILM on 302 million tokens, this work used 7.7% of the RAM, 25.4% of the CPU time, and 14.0% of the wall time. Compared with IRSTLM, this work used 16.6% of the RAM, 16.4% of the CPU time, 9.0% of the wall time. MITLM is admirably fast; in fact, it is slightly faster in terms of CPU time. However, this work is faster in terms of wall time due to threading.

3.6.4 Scaling

Estimating an upruned 5–gram model (Table 3.1) on 126 billion tokens took 123 GB RAM, 5.4 CPU days, and 2.8 days wall time on a machine with 140 GB RAM and six hard drives in a RAID5 configuration (sustained read: 405 MB/s). A summary of Google's results appears in Table 3.2. Using Chapter 4, the



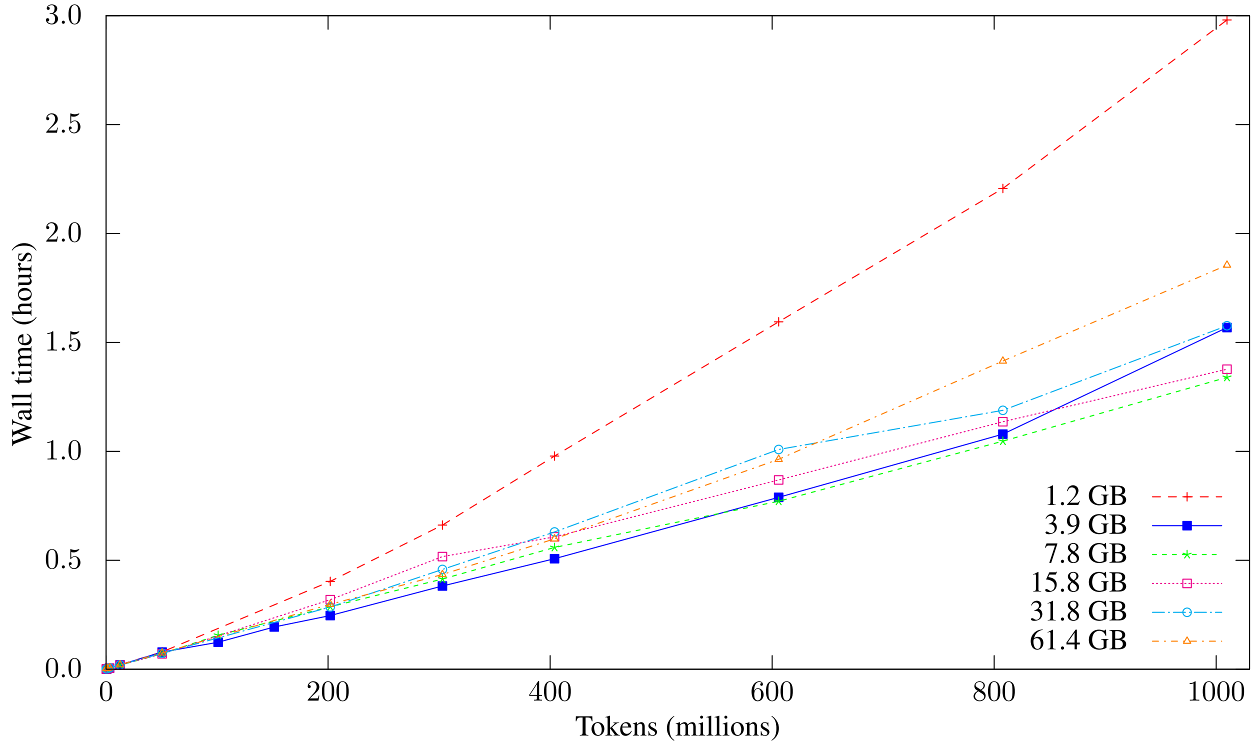


Figure 3.5: Performance of this work with various memory settings. Curves are labeled by their empirical peak memory usage. CPU time is largely unaffected by the memory setting on small corpora. When memory is particularly tight (1.2 GB), wall time increases because lazy merge sort is no longer feasible.

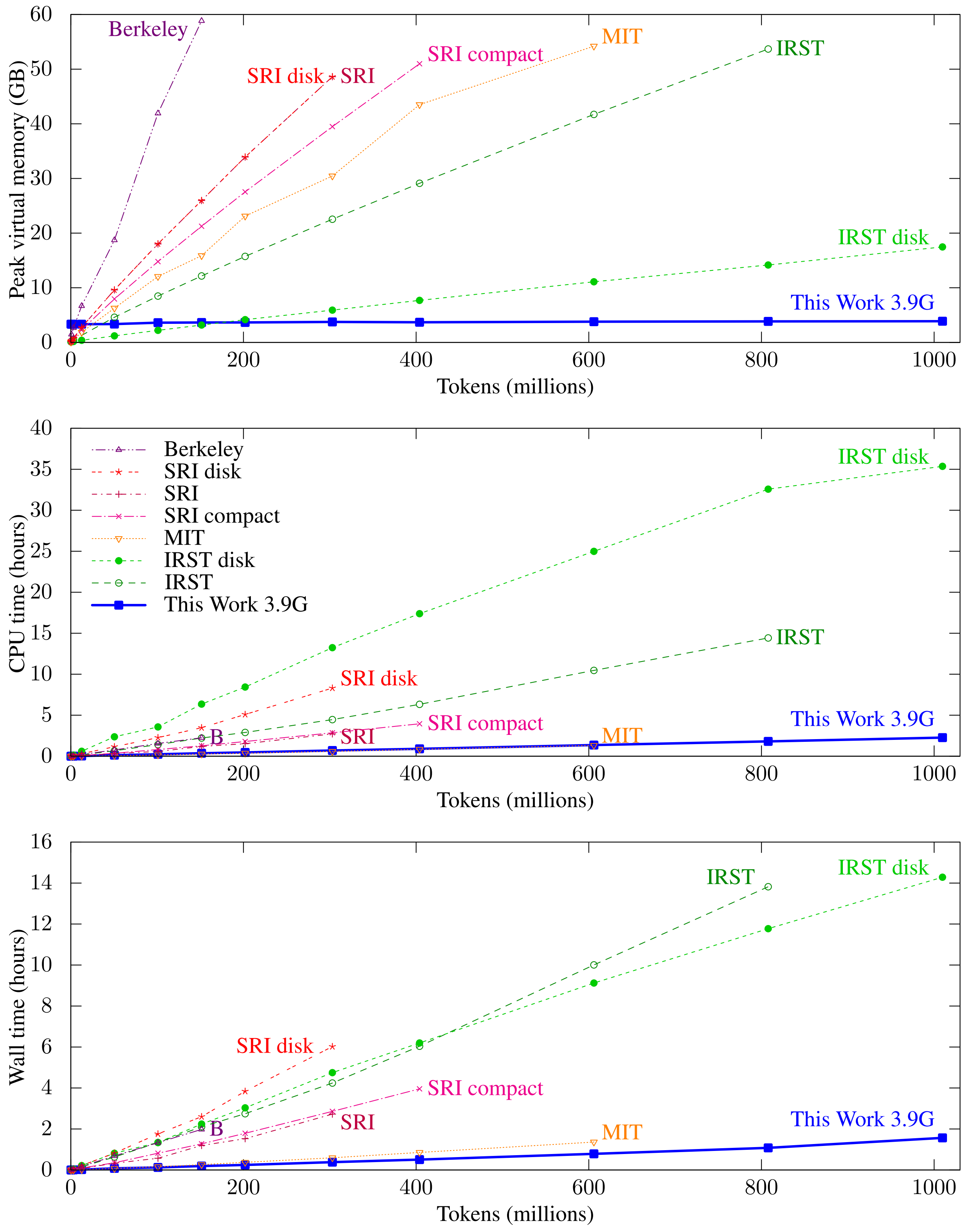


Figure 3.6: Performance of various toolkits. Experiments were stopped before they ran out memory. SRILM's default and disk-based pipeline use the same RAM for unpruned models.

1	2	3	4	5
393M	3,775M	17,629M	39,919M	59,794M

Table 3.1: Counts of unique n-grams (in millions) for the 5 orders in the large language model.

	Tokens	Smoothing	Machines	Days	Year
This Work	126 billion	Kneser-Ney	**1**	2.8	2013
Google	31 billion	Stupid Backoff	400	0.3	2007
	31 billion	Kneser-Ney	400	2	2007
	230 billion	Kneser-Ney	?	?	2013
	1800 billion	Stupid Backoff	1500	1	2007

Table 3.2: Large-scale results from this work and Google (Brants et al., 2007; Chelba and Schalkwyk, 2013). Chelba and Schalkwyk (2013) did not report the cost of estimating the language model with 230 billion tokens. Comparisons are somewhat unfair due to differences in hardware and data.

large model was quantized (Whittaker and Raj, 2001) to 10 bits and compressed to 643 GB then copied to a machine with 1 TB RAM. Lossy compression methods (Guthrie and Hepple, 2010; Talbot and Osborne, 2007) and distributed language models (Brants et al., 2007) could reduce hardware requirements. The large model was added to an existing machine translation system.

Built and described by Durrani et al. (2013d), the baseline phrase-based Moses (Koehn et al., 2007) system was trained on all the data provided by the evaluation (Bojar et al., 2013a), totaling 7 billion tokens of English. Using SRILM (Stolcke, 2002), 5–gram interpolated modified Kneser-Ney language models were separately estimated on each year of the news crawl data, the news commentary data, each source in the English Gigaword corpus (Parker et al., 2011), and the English sides of each parallel corpus provided by the evaluation (Bojar et al., 2013a). Default SRILM pruning was used, so singletons were removed for trigrams and above. These models were then linearly interpolated with weights tuned on the 2011 test set, yielding the baseline language model. Feature weights were tuned with PRO (Hopkins and May, 2011) for Czech–English and batch MIRA (Cherry and Foster, 2012) for French–English and Spanish–English because these worked best on the 2012 test set.

The large language model was added to the baseline as a separate feature.[12] Feature weights were retuned using the same tuning algorithms. Due to time limits on access to machines with 1 TB RAM, the large language model was only used in Czech–English, French–English, and Spanish–English submissions to the 2013 Workshop on Machine Translation. The additional language model feature and retuning are the only difference between this work and the baseline. In particular, only parallel data provided by the evaluation was used; Google presumably used more data.

Results from the 2013 Workshop on Machine Translation are shown in Table 3.3. On the official metric (Bojar et al., 2013a), which is based on human comparisons against all participants, using a large language model significantly improves over both the baseline and Google. In direct pairwise comparisons (not shown), improvement over the baseline is significant ($> 99\%$ confidence) in all three language pairs. Pairwise improvement over Google is significant in French–English ($> 99\%$ confidence) and insignificant for Czech–English and Spanish–English (Bojar et al., 2013a).

That large language models improve translation is not a new result (Brants et al., 2007). However, Brants et al. (2007) recommended stupid backoff. Osborne (2013) estimated a stupid backoff language model on the same ClueWeb09 data, lossily compressed it with RandLM (Talbot and Osborne, 2007), and reported no

[12]Experiments with Czech–English found that a separate feature works better, in terms of BLEU (Papineni et al., 2002) on the 2012 test set (Callison-Burch et al., 2012), than interpolation.

Czech–English (11 total participants)

	Human		Automatic		
System	Rank	Score	BLEU	TER	METEOR
This Work	1	0.607	28.16	54.32	63.32
Google	2–3	0.582	27.11	53.60	62.71
Baseline	3–5	0.562	27.38	54.99	62.64

French–English (13 total participants)

	Human		Automatic		
System	Rank	Score	BLEU	TER	METEOR
This Work	1	0.638	33.37	49.81	66.67
Google	2–3	0.591	32.62	49.44	66.23
Baseline	2–3	0.604	32.57	50.42	66.28

Spanish–English (12 total participants)

	Human		Automatic		
System	Rank	Score	BLEU	TER	METEOR
This Work	1	0.624	32.55	50.34	66.83
Google	2	0.595	33.65	48.17	66.98
Baseline	3–5	0.570	31.76	50.90	66.29

Table 3.3: Results from the 2013 Workshop on Machine Translation (Bojar et al., 2013a). Also shown are uncased BLEU (Papineni et al., 2002), TER (Snover et al., 2006), and METEOR 1.4 (Denkowski and Lavie, 2011).

improvement in a different machine translation system.

Output Case Studies

To illustrate the impact of the large language model, two sentences were cherry picked from the 3000-sentence official test set. In the first example, human judges preferred the large language model over the baseline. In the second example, human judges preferred the baseline (Bojar et al., 2013a).

The large system reasonably translated a Czech word that the baseline had negated.

Source	Za tím se skrývá označení, které je v Číně **zcela** oficiálně registrováno, avšak s původní firmou nemá nic společného.
Reference	Behind this name hides a **fully** officially registered brand in China, however, one that has nothing whatsoever to do with the original companies.
This Work	It is a label that is **completely** officially registered in China, but has nothing to do with the original company.
Baseline	It is the designation, which is **not** officially registered in China, but has nothing to do with the original company.

The Czech word "zcela" in this case translates as "fully". It is frequently used to qualify negative sentences, but does not itself indicate negation.[13] This led to an erroneous phrase pair that translates "zcela" as "not". Nonetheless, the translation model correctly favors "completely" (Table 3.4).

The baseline language model favored the negated translation while the large language model slightly favors the better translation (Table 3.5). This happened, in part, because the baseline language model train-

[13]Thanks to Czech speaker Bojar (2013) for this explanation.

ing data did not contain "completely officially", so the model backed off to a unigram in order to compute $p\,(\text{officially} \mid \text{label that is completely})$. In contrast, the large language model had seen the trigram "is completely officially" and gave it a higher probability as a result.

	completely	not
Score in This Work	-0.26	-1.08
Score in Baseline	-0.28	-1.09

Table 3.4: Translation model scores for "zcela" after computing the dot product with feature weights. Only the relevant two (of 3,833) ways to translate "zcela" are shown. Scores are slightly different because the system was retuned to accomodate the large language model.

	This Work's Translation	Baseline's Translation
Large Model	-38.05	-38.70
Baseline Model	-37.74	-34.26
Score in This Work	-4.79	-4.52
Score in Baseline	-4.60	-4.17

Table 3.5: Sentence-level language model scores ($\log_{10}$ probability) for the first example. Scores are also shown after weights are applied. Both models were used as features in the large system.

Lanuage models run the risk of favoring common words over those that reflect the input:

Source	Voici l'histoire d'un **lien** qui **franchit** bien plus qu'une distance de 8,733 kilomètres.
Reference	The history of a **link** that **overcomes** far more than a distance of 8,733 kilometres.
This Work	This is the story of a **relationship** that **is** much more than a distance of 8,733 kilometers.
Baseline	This is the story of a **link** that **passes** much more than a distance of 8,733 kilometers.

The French verb "franchir" (of which "franchit" is a morpological variant) means to cross or to overcome. In this case, it was erroneously translated as "is". Such errors happen easily when phrases are extracted from parallel text (Koehn et al., 2007; Koehn et al., 2003). For example, the French sentence "De plus, la circulation routière, dans l'Europe d'aujourd'hui, franchit largement les frontières." translates as "In addition, traffic in Europe today is, to a very large extent, transboundary." in the Europarl corpus (Koehn, 2005). In some sense, aligning "is" to "franchit" is correct because these are the main verbs in the sentence, though "transboundary" carries its meaning. As with the previous example, the phrase scores favor the better translation (Table 3.6).

As shown in Table 3.7, both language models prefer the erroneous translation. Their preference is reasonable given that "is" is more frequent than "passes". The difference is that the large language model makes a stronger distinction and overpowers the translation model.

3.7 Summary

This chapter has shown how to accurately and relatively inexpensively estimate language models with interpolated modified Kneser-Ney smoothing. Disk-based streaming and sorting algorithms make estimation fast while using user-specified RAM. Somewhat surprisingly, it is faster than existing memory-based toolkits. The difference is that memory-based toolkits typically perform data-structure lookups every time they access an n-gram while, in this work, relevant records are immediately found in the streams. It is hoped that the community will find the open-source implementation, dubbed `lmplz`, useful and make fuller use of the large amounts of data available on the web.

Future directions include an analogous algorithm to interpolate separately trained language models,

	is	passes
Score in This Work	-1.32	-0.66
Score in Baseline	-1.30	-0.67

Table 3.6: Translation model scores for "franchit" after computing the dot product with feature weights. Only the relevant two (of 356) ways are shown. Scores are slightly different because the system was retuned to accomodate the large language model.

	This Work's Translation	Baseline's Translation
Large Model	-36.22	-45.14
Baseline Model	-32.06	-37.46
Score in This Work	-4.23	-5.07
Score in Baseline	-3.91	-4.57

Table 3.7: Sentence-level language model scores ($\log_{10}$ probability) for the second example. Scores are also shown after weights are applied.

pruning, different smoothing methods, paritioning the set of n-grams for parallelization, optional compression, and other applications of the general streaming framework.

Chapter 4

Querying Language Models[1]

A place for everything and everything in its place.
–Isabella Mary Beeton, *The Book of Household Management*

Have a place for everything and keep the thing somewhere else; this is not advice, it is merely custom.
–Mark Twain

This chapter is about efficiently querying backoff-smoothed N-gram language models (Katz, 1987), including interpolated modified Kneser-Ney models from Chapter 3. Applications frequently query the language model in their search for high-scoring derivations (Section 2.4); for example, a machine translation system makes millions of queries to translate an average sentence (Section 5.6). Query speed is therefore important to overall system performance, a relationship borne out in the experiments. Moreover, while queries are not random (the basis for Chapter 5), they are also not sequential, so the disk-based techniques used in Chapter 3 do not efficiently apply to queries. Thus, language models are typically stored in RAM (Section 2.3), where space is a constant concern. The goals in this chapter are to increase query speed and decrease memory usage while accurately reporting the language model's probabilities.

Both Chapter 3 and this chapter support the standard ARPA format for compatibility with many other language model toolkits. The two can also be combined to build a query-time representation directly from a corpus, as was done with the experiments in Chapter 3. While Chapters 5, 6, and 7 could be applied independently of language model representation, they are made more efficient by integrating with the language model representation. For example, Chapter 6 stores an additional value alongside probability and backoff.

4.1 Introduction

Language model probability queries take the form $p(w_n \mid w_1^{n-1})$ where w_1^n is an n-gram. Backoff-smoothed models (Katz, 1987) estimate this probability according to the recursive equation

$$p(w_n \mid w_1^{n-1}) = \begin{cases} p(w_n \mid w_1^{n-1}) & \text{if } w_1^n \text{ appears in the model} \\ b(w_1^{n-1})p(w_n \mid w_2^n) & \text{otherwise} \end{cases}$$

where probability p and backoffs b were estimated from text by some smoothing method. Unraveling the recursive equation yields an equivalent procedure: match as much context as possible then charge backoffs

[1]An earlier version of this work was published as Heafield (2011).

b for contexts that were not matched.

$$f = \min\{i : w_i^n \text{ appears in the model}\}$$

$$p(w_n \mid w_1^{n-1}) = p(w_n \mid w_f^{n-1}) \prod_{i=1}^{f-1} b(w_i^{n-1})$$

The goal is to store a large and sparse set of n-grams, along with their probabilities and backoffs, so that executing this procedure is efficient.

The most common data structure is a reverse trie, shown in Figure 4.1 (Clarkson and Rosenfeld, 1997). Nodes in the trie correspond to n-grams in the language model. When queried for $p(w_n \mid w_1^{n-1})$, the language model walks down the tree, visiting nodes $w_n^n, w_{n-1}^n, \ldots$ until it stops at w_f^n because all words have been matched ($f = 1$) or because the model does not contain w_{f-1}^n. The model then reads $p(w_n \mid w_f^{n-1})$ and, if necessary, performs a second walk through the trie to retrieve backoff values $\{b(w_i^{n-1})\}_{i=1}^{f}$, which are all found on the path $w_{n-1}^{n-1}, w_{n-2}^{n-1}, \ldots$. This path may also reach a dead end; backoff is implicitly 1 if there is no entry in the model.

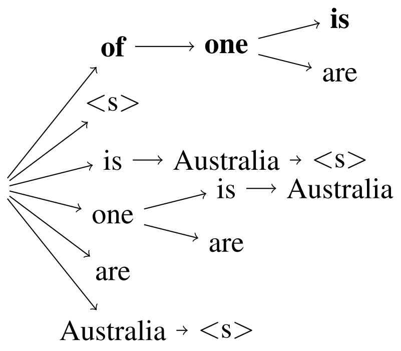


Figure 4.1: Lookup of "is one of" in a reverse trie. Given query w_1^n, the last word w_n is looked up first, so that w_f^n is found efficiently.

Many have implemented language model querying with various data structures:

SRILM 1.5.12 (Stolcke, 2002) is a popular toolkit based on tries.

IRSTLM 5.60.02 (Federico et al., 2008) is a trie implementation based on sorted arrays. It is designed to reduce memory consumption relative to SRILM.

MITLM 0.4 (Hsu and Glass, 2008) is mostly designed for accurate model estimation, but can also compute probability using a trie.

TPT Germann et al. (2009) describe tries with better locality properties, but no implementation is available.

RandLM 0.2 (Talbot and Osborne, 2007) uses randomized data structures based on Bloom filters (Bloom, 1970) to conserve memory.

Google Talbot and Brants (2008) use minimal perfect hashing and fingerprints to save space.[2]

[2]Google is excluded from comparisons because no implementation is available and Talbot and Brants (2008) only reported memory consumption, which is higher than whith ShefLM.

ShefLM revision 13974bc (Guthrie and Hepple, 2010) implements improved minimal perfect hashing techniques (Belazzougui et al., 2008) that use less memory than Google.

BerkeleyLM revision 152 (Pauls and Klein, 2011) implements tries based on hash tables and sorted arrays.

Packages are further described in Section 4.3. This work presents two data structures, collectively dubbed KenLM, that substantially outperform all baselines on query speed. In addition to being faster than all baselines, the trie data structure uses less RAM than lossless baselines.

4.2 Data Structures

Sparse mapping is a key subproblem of language model storage. One option is to map n-grams directly to their probability and backoff. The first data structure, dubbed probing, applies linear probing hash tables and is optimized for speed. Another option is to arrange n-grams into a trie. In a trie, each node contains a sparse map from vocabulary identifiers to child nodes. The second data structure performs this mapping using a variation on binary search through a sorted array of records.

4.2.1 Hash Tables and Probing

Hash tables are a common sparse mapping technique used by SRILM's default. Keys to the table are hashed, using for example MurmurHash (Appleby, 2012), to integers evenly distributed over a large range. This range is collapsed to a number of buckets, typically by taking the hash modulo the number of buckets. Entries landing in the same bucket are said to collide.

Chaining is a popular technique for collision resolution shown in Figure 4.2. Colliding entries are stored in a linked list associated with their ideal bucket. The problem with chaining is that a successful lookup requires at least two random memory reads: one to index the array of buckets and another to follow the linked list pointer. Moreover, pointers consume at least 8 bytes of memory per entry on a 64-bit machine.

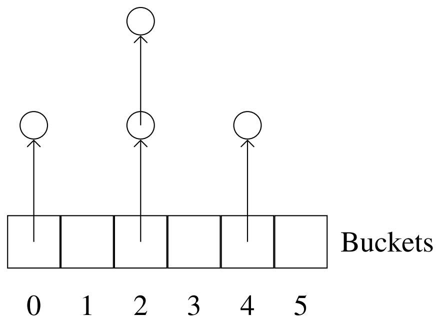


Figure 4.2: A chaining hash table with six buckets (squares) and four entries (circles). Each bucket contains a possibly-empty linked list of entries.

Linear probing hash tables (Knuth, 1963) have lower memory overhead when entries are small and, in the vast majority of cases, require only one random memory access per lookup. The data structure is an array of buckets. Each bucket contains exactly one entry or is empty. Adding an element consists of finding the ideal bucket (hash modulo the number of buckets), scanning forward in the array until an empty bucket is found, and placing the new entry there. Therefore, non-empty buckets contain an entry belonging to them or to a preceding bucket where a collision occurred. Searching a probing hash table consists of hashing the key, indexing the corresponding bucket, and scanning buckets until a matching key is found or an empty

		Bigrams		
Words	**Ideal**	**Hash**	**log p**	**log b**
iran is	0	`959e48455f4a2e90`	-1.7	-0.4
		`0`	0	0
is one	2	`186a7caef34acf16`	-2.0	-0.9
one of	2	`ac66610314db8dac`	-1.4	-0.6
<s> iran	4	`f0ae9c2442c6920e`	-3.3	-1.2
		`0`	0	0

Table 4.1: Memory layout of the linear probing hash table with six buckets and four bigram entries. The column "ideal" is the hash modulo the number of buckets, where each entry would ideally reside before collision resolution. The boxed area is actually stored in memory as an array. Hash values are 64-bit integers, shown here in base 16.

bucket is encountered, in which case the key does not exist in the table. The vast majority of forward scans do not wrap around to the beginning of the array, so almost all queries entail one random memory access followed by a sequential scan.

Linear probing hash tables must have more buckets than entries, or else an empty bucket will never be found. The ratio of buckets to entries is controlled by space multiplier $m > 1$. As the name implies, space is $O(m)$ and linear in the number of entries. The fraction of buckets that are empty is $\frac{m-1}{m}$, so average lookup time is

$$O\left(\frac{m}{m-1}\right)$$

and, theoretically, constant in the number of entries. In practice, lookup is faster for smaller hash tables due to caching.

When keys are longer than 64 bits, space is conserved by replacing the keys with their 64-bit hashes. With a good hash function, collisions of the full 64-bit hash are exceedingly rare: for the model tested, one in 266 billion queries will falsely find a key not present. Collisions between two keys in the table can be identified at model building time. Further, the special hash 0 suffices to flag empty buckets.

The probing data structure is a rather straightforward application of linear probing hash tables to store N-gram language models. Unigram lookup is dense so an array of probability and backoff values is used. For $2 \leq n \leq N$, a hash table maps from n-grams to their respective probability and backoff[3]. An example appears in Table 4.1. Vocabulary lookup is a hash table mapping from word to vocabulary index. In all cases, the key is collapsed to its 64-bit hash. Given counts c_1^N, where e.g. c_1 is the vocabulary size, total memory consumption, in bits, is

$$(96m + 64)c_1 + 128m \sum_{n=2}^{N-1} c_n + 96mc_N.$$

The probing data structure places all n-grams of the same order into a single giant hash table. This differs from other implementations (Stolcke, 2002; Pauls and Klein, 2011) that use hash tables as nodes in a trie, as explained in the next section. As implemented here, queries can jump directly to any n-gram of any length with a single lookup, a property shared with ShefLM. Backoff retrieval can be therefore be slightly optimized by directly looking up $b(w_1^{n-1})$ without retrieving $b(w_{n-1}^{n-1})$, as would be required with a trie.

[3] N-grams do not have backoff so none is stored.

4.2.2 Sorted Arrays and Trie

```
lower $\leftarrow 0$
upper $\leftarrow |A| - 1$
while lower $\leq$ upper do
    pivot $\leftarrow \left\lfloor \frac{\text{key} - A[\text{lower}]}{A[\text{upper}] - A[\text{lower}]}(\text{upper} - \text{lower}) \right\rfloor + \text{lower}$
    if $A[\text{pivot}] > \text{key}$ then
        upper $\leftarrow$ pivot
    else if $A[\text{pivot}] < \text{key}$ then
        lower $\leftarrow$ pivot
    else
        return pivot
    end
end
return Not found
```

Algorithm 3: Interpolation search for a key in an array A. If the array contains repeated entries, the algorithm is slightly modified to avoid division by zero.

Sorted arrays store key-value pairs in an array sorted by key, incurring no space overhead. SRILM's compact variant, IRSTLM, MITLM, and BerkeleyLM's sorted variant are all based on this technique. Given a sorted array A, these other packages use binary search to find keys in $O(\log |A|)$ time. This can be reduced to $O(\log \log |A|)$ time by evenly distributing vocabulary identifiers (e.g. in order of 64-bit hashes of words) then applying interpolation search [4] (Perl et al., 1978). Shown in Algorithm 3, interpolation search formalizes the notion that one opens a dictionary near the end to find the word "zebra." Interpolation search is a form of binary search with better estimates informed by the uniform key distribution.

If the key distribution's range is also known (i.e. vocabulary identifiers range from zero to the number of words minus one), then interpolation search can use this information instead of reading $A[0]$ and $A[|A|-1]$; this optimization alone led to a 24% speed improvement. The improvement is due to the cost of bit-level reads and avoiding reads that may fall in different virtual memory pages. Virtual memory pages are typically 4096 bytes in size.

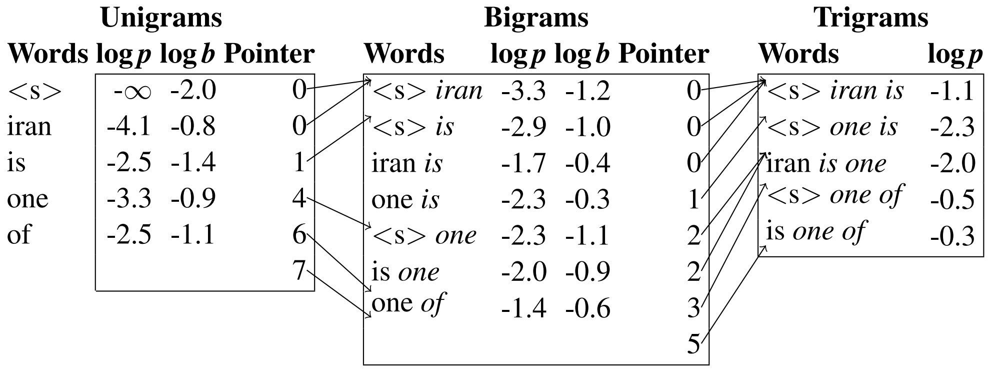
Vocabulary lookup is a sorted array of 64-bit word hashes. The index in this array is the vocabulary identifier. This has the effect of randomly permuting vocabulary identifiers, meeting the requirements of interpolation search when vocabulary identifiers are used as keys.

While sorted arrays could be used to implement the same data structure as with the probing model, effectively making $m = 1$, this implementation was abandoned because it was empirically slower and larger than a trie implementation. The trie implementation is based on the popular reverse trie, in which the last word of an n-gram is looked up first. Reverse tries are also implemented by SRILM, IRSTLM's inverted variant, and BerkeleyLM except for the scrolling variant. Figure 4.1 shows a reverse trie at a high level. Nodes in the trie are based on arrays sorted by vocabulary identifier.

When the language model arrives at some node w_i^n, it needs to know where to find the leftward extensions vw_i^n in order to continue looking up more context. SRILM stores 64-bit pointers, but this is costly in terms of memory. Instead, this work adopts a technique introduced by Clarkson and Rosenfeld (1997) and subsequently implemented by IRSTLM and BerkeleyLM's compressed option. Pointers are actually array indices. All n-grams of the same length are stored in a large array (Figure 4.3). These arrays are sorted in

[4]Not to be confused with interpolating probabilities, this search interpolates vocabulary indices.

suffix order: w_n is the primary sorting key, w_{n-1} is secondary, etc. Suffix order was also used to estimate language models (Figure 3.3). This sorting order ensures that, for any entry w_i^n, its leftward extensions $\{vw_i^n\}$ are stored consecutively. The entry w_i^n stores the index at which $\{vw_i^n\}$ begins. The entry that follows w_i^n indicates where the next block begins, also indicating where the block of $\{vw_i^n\}$ ends.



Unigrams

Words	log *p*	log *b*	Pointer
<s>	$-\infty$	-2.0	0
iran	-4.1	-0.8	0
is	-2.5	-1.4	1
one	-3.3	-0.9	4
of	-2.5	-1.1	6
			7

Bigrams

Words	log *p*	log *b*	Pointer
<s> *iran*	-3.3	-1.2	0
<s> *is*	-2.9	-1.0	0
iran *is*	-1.7	-0.4	0
one *is*	-2.3	-0.3	1
<s> *one*	-2.3	-1.1	2
is *one*	-2.0	-0.9	2
one *of*	-1.4	-0.6	3
			5

Trigrams

Words	log *p*
<s> *iran is*	-1.1
<s> *one is*	-2.3
iran *is one*	-2.0
<s> *one of*	-0.5
is *one of*	-0.3

Figure 4.3: A trigram language model stored in a trie. Pointers stored in the table of n-grams are indices into the array of $(n+1)$-grams. The *italic* words are implicitly encoded by the pointers in the trie. Block length is determined by reading the following record. To support this operation for the last valid record, a dummy record with a pointer has been added.

When SRILM (Stolcke, 2002) prunes a model, it sometimes removes n-grams but not $(n+1)$–grams that extend it to the left. This is due to count thresholding after counts have been adjusted for Kneser-Ney smoothing (Kneser and Ney, 1995) as described in Section 3.3.2. In a model built with default settings, 1.2% of $n+1$–grams were missing their n-gram suffix. Such omissions cause problems for reverse trie implementations, including SRILM itself, because there is no n-gram node that points to their records in the array of $n+1$–grams. In this work, the issue is resolved by hallucinating the necessary n-grams. Hallucinated n-grams have probability calculated according to the normal backoff algorithm and themselves have backoff equal to one (the implicit default value for missing entries), so the probabilities reported by the model remain unchanged. However, the language model also reports the value of f when it uses $p(w_n \mid w_f^{n-1})$ in response to a query for $p(w_n \mid w_1^{n-1})$ as explained in the introduction (Section 4.1). Decoders may use f as a feature and the value of f will decrease when a hallucinated entry is used.

Optional Quantization

Floating point values may be stored in the trie exactly, using 31 bits for non-positive log probability and 32 bits for log backoff.[5] To conserve memory at the expense of accuracy, values may be quantized using Q_p bits per probability and Q_b bits per backoff[6]. This work allows any number of bits from 2 to 25, unlike IRSTLM (8 bits). Quantized values are selected with the binning method (Federico and Bertoldi, 2006) that sorts values, divides into equally sized bins, and averages within each bin. The cost of storing these averages, in bits, is

$$32(N-1)2^{Q_p} + 32(N-2)2^{Q_b}$$

[5]Section 3.4 proved that backoff is at most one in unpruned language models with interpolated modified Kneser-Ney smoothing. In that case, log backoff is always non-positive, so the sign bit could similarly be omitted. However, the intent is to support general smoothing algorithms and pruning.

[6]Two values are allocated to zero backoff to support right state minimization in Section 5.3.2. That leaves 2^{Q_p} probabilities and $2^{Q_b} - 2$ non-zero backoffs.

Chapter 6 shows how to add another floating-point value to improve search accuracy or collapse probability and backoff into a single value, decreasing search accuracy. These floating-point values can be quantized in the same way.

Optional Offset Compression

Pointers stored by the trie are in increasing order, comprising a sorted array. Sorted arrays can be compressed by removing the first bit and storing the offset at which the first bit changed to one. More bits can be chopped in this way, using an array of 2^R offsets to store where the leading R bits roll over. For example, if $R = 4$, then pointers are stored without their 4 most significant bits. An array of size 16 stores offsets where those bits roll over. Thus, the 7th entry in the array contains the offset where the leading bits start to be 0111, which is 7 in binary. This is a simplified version of the compression technique introduced by Raj and Whittaker (2003). Use of this compression is optional. When enabled, the language model chooses R_n separately for each order n so as to minimize the total size of the offset and trie tables. Lookup time is linear in R_n due to binary search through the array of 2^{R_n} offsets, so the user can set a lower value if desired.

Total Storage Cost

Unigram records store probability, backoff, and an index in the bigram table. Entries for $2 \leq n < N$ store a vocabulary identifier, probability, backoff, and an index into the $n + 1$–gram table. The highest-order n-gram array omits backoff and the index, since these are not applicable. Values in the trie are minimally sized at the bit level, improving memory consumption over trie implementations in SRILM, IRSTLM, and BerkeleyLM. Given n-gram counts $\{c_n\}_{n=1}^{N}$, each vocabulary identifier costs $\lceil \log_2 c_1 \rceil$ bits while indices into the table of n-grams cost $\lceil \log_2 c_n \rceil$ bits each, minus any offset compression.

Because there are comparatively few unigrams, they are stored byte-aligned and unquantized, making every query faster. Unigrams also have 64-bit overhead for vocabulary lookup. Using c_n to denote the number of n-grams, total memory consumption of the trie, in bits, is

$$\begin{aligned} &(32 + 32 + 64 + 64)c_1 + \\ \sum_{n=2}^{N-1} &(\lceil \log_2 c_1 \rceil + Q_p + Q_b + \lceil \log_2 c_{n+1} \rceil - R_n)c_n + \\ &(\lceil \log_2 c_1 \rceil + Q_p)c_N \end{aligned}$$

plus quantization and offset tables, if used. The size of the trie is particularly sensitive to $\lceil \log_2 c_1 \rceil$, so vocabulary filtering is quite effective at reducing model size.

4.3 Related Work

One straightforward way to fit a language model in memory is to store fewer n-grams. Reducing N (the order of the model) and pruning n-grams with low counts or probabilities (Stolcke, 2002; Moore and Quirk, 2009; Chelba et al., 2010) are two common, but lossy, methods. Careful data selection (Moore and Lewis, 2010) removes low-quality training data and actually leads to smaller and better models. Filtering (Stolcke, 2002; Heafield and Lavie, 2010) removes n-grams that a decoder will not query because no sentence in the corpus can produce all words in the n-gram. Sentence-filtered language models incur substantially more disk operations and are rarely used. For a competitive translation system, filtering to the evaluation's test set yielded a 36% reduction in the number of n-grams. Finally, some work (Federmann, 2007; Talbot and Osborne, 2007) stores language models on a network, splitting the memory requirement over multiple machines but incurring the cost of network latency. Once the language model has been optionally pruned, filtered, and distributed, it still remains to efficiently represent the model in memory.

SRILM (Stolcke, 2002) is widely used within academia. It is generally considered to be fast (Pauls and Klein, 2011), with a default implementation based on hash tables within each trie node. Each trie node is individually allocated and full 64-bit pointers are used to find them, wasting memory. The compact variant uses sorted arrays instead of hash tables within each node, saving some memory, but still stores full 64-bit pointers. With some minor API changes, namely returning the length of the n-gram matched, it could also be faster—though this would be at the expense of an optimization explained in Section 5.3.2. The probing model was designed to improve upon SRILM by using linear probing hash tables (though not arranged in a trie), allocating memory all at once (eliminating the need for full pointers), and being easy to compile.

IRSTLM (Federico et al., 2008) is an open-source toolkit for building and querying language models. The developers aimed to reduce memory consumption at the expense of time. Their default variant implements a forward trie, in which words are looked up in their natural left-to-right order. However, their inverted variant implements a reverse trie using less CPU and the same amount of memory.[7] Each trie node contains a sorted array of entries and they use binary search. Compared with SRILM, IRSTLM adds several features: lower memory consumption, a binary file format with memory mapping, caching to increase speed, and quantization. The trie implementation is designed to improve upon IRSTLM using a reverse trie with improved search, bit level packing, and stateful queries. IRSTLM's quantized variant is the inspiration for quantization in this work.

Lossily compressed models RandLM (Talbot and Osborne, 2007) and ShefLM (Guthrie and Hepple, 2010) offer better memory consumption at the expense of CPU and accuracy. These enable much larger models in memory, compensating for lost accuracy. Typical data structures are generalized Bloom filters that guarantee a customizable probability of returning the correct answer. Minimal perfect hashing is used to find the index at which a quantized probability and possibly backoff are stored. These models generally outperform KenLM's memory consumption but are much slower, even when cached.

BerkeleyLM (Pauls and Klein, 2011) is included for completeness despite its release after the release of this work. Like SRILM, it implements tries using hash tables and sorted arrays. It also offers the compressed variant that uses standard block compression (gzip), which is rather slow. It does contribute an optional direct-mapped cache designed to quickly remember probabilities of repeatedly queried n-grams. This work does not use caching and is still faster on cache-friendly queries.

4.4 Threading and Memory Mapping

Support for threading is trivial because the data structures are read-only and uncached. Only IRSTLM does not support threading. Memory mapping also allows multiple processes on the same machine to share the language model.

Memory mapping is an efficient operating system mechanism where a file and part of RAM have the same content. Along with IRSTLM and TPT, the binary format is memory mapped. This is especially effective at reducing load time, since raw bytes are read directly to memory—or, as happens with repeatedly used models, are already in the disk cache.

On Linux, memory mapping can be configured to aggressively load the file from disk (with the flag `MAP_POPULATE`) or lazily load sections from disk as they are accessed. Lazy memory mapping has the advantage that it uses less physical memory (because some pieces of the file are never accessed), but is generally slow because queries against unloaded portions must wait for the disk. This is especially bad with the probing data structure because it is based on hashing and performs random lookups. However, the

[7]Forward tries are faster to build with IRSTLM and can efficiently return a list of rightward extensions. This functionality can be useful in speech recognition applications, but is not used by the decoding algorithms under consideration.

probing data structure is not intended to be used in low-memory scenarios. The trie data structure uses less memory and has better locality. It partitions storage by n-gram length, so walking the trie reads N disjoint pieces of the file. TPT has theoretically better locality because it stores n-grams near their suffixes, thereby placing reads for a single query in the same or adjacent areas or the file.

Agressive loading is the default in this work, though lazy loading is an option. Experiments do not cover models larger than physical memory because TPT is unreleased, factors such as disk speed are hard to replicate, and in such situations the recommendation is to switch to a more compact representation, such as RandLM or ShefLM. In all of the experiments, the binary file (whether mapped or, in the case of most other packages, interpreted) is loaded into the disk cache in advance, thereby putting the disk cache in a consistent state. This also ensures that even lazy memory mapping will not wait for the disk.

4.5 Experiments

This section measures performance on shared tasks in order of increasing complexity: sparse lookups, evaluating perplexity of a large file, and translation with Moses (Koehn et al., 2007). The test machine has two Intel Xeon E5410 processors totaling eight cores, 32 GB RAM, and four Seagate Barracuda disks in software RAID 0 running Linux 2.6.18. Experimental results also incorporate two optimizations described in Chapter 5, right state minimization (Section 5.3.3) and backoff representation (Section 5.4), because these were implemented at the time but logically belong with left state optimization.

4.5.1 Sparse Lookup

Figure 4.4 compares the custom linear probing hash table with chaining implementations `hash_set` provided by GCC and `unordered_set` from Boost (James, 2005). It also compares interpolation search with ordinary `binary_search` and the standard C++ `set` based on red-black trees. Each data structure was populated with 64-bit integers sampled uniformly without replacement. They were then queried 20 million times: 10 million times with uniformly sampled valid keys and 10 million times with uniformly sampled invalid keys, both with replacement. The same entries and queries were used for each data structure. Time includes all queries but excludes random number generation and data structure population.

Performance dips as each data structure outgrows the processor's 12 MB second-level cache. The number of n-grams ranges from a few million to hundreds of billions, so when n-grams are stored in a hash table (as in the case with the probing data structure), the most relevant values are on the right size of the graph, where linear probing and interpolation search win.

Among hash tables, indicated by shapes, linear probing is initially slower but converges to 43% faster than both `unordered_set` and `hash_set`. The linear probing hash table is faster on large data because it does only one random memory access per query while chaining operations generally perform two random memory accesses. It also uses less memory, with 8 bytes of overhead per entry (16-byte entries with $m = 1.5$); chaining implementations `hash_set` and `unordered_set` require at least 8 bytes of overhead per entry to store pointers.

Among sorted lookup techniques, interpolation search is initially the slowest because it has a more expensive pivot function. With larger tables (above 4096 entries), interpolation search is faster because it converges in fewer iterations. This suggests a strategy: run interpolation search until the range narrows to 4096 or fewer entries, then switch to `binary_search`. However, reads in the trie data structure are more expensive due to bit-level packing, so it is faster to use interpolation search the entire time. Memory usage of interpolation search is the same as with `binary_search` and lower than with `set`.

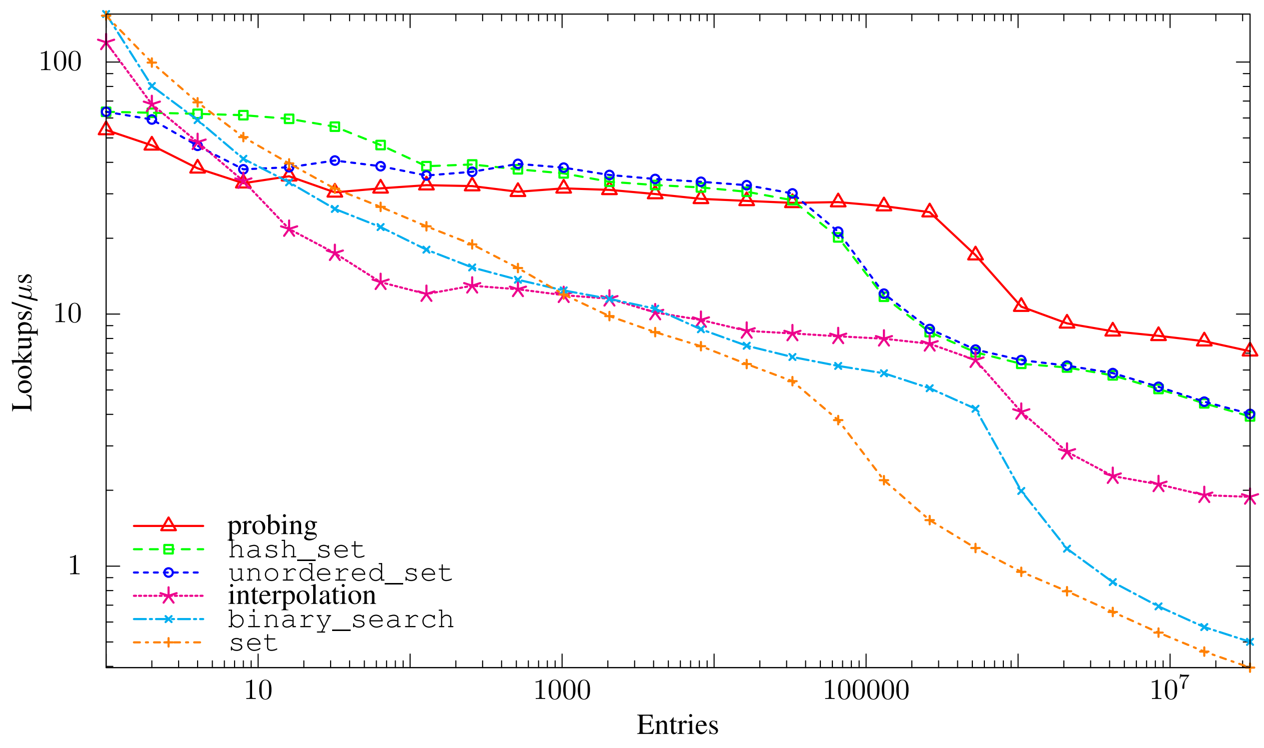


Figure 4.4: Speed in lookups per microsecond by data structure and number of 64-bit entries.

4.5.2 Perplexity

The perplexity and translation tasks use a 5–gram English language model estimated on 834 million tokens from Europarl version 6 (Koehn, 2005) and the 2011 Workshop on Machine Translation News Crawl corpus with duplicate lines removed (Callison-Burch et al., 2011). SRILM (Stolcke, 2002) was used to esimate the model[8] with interpolated modified Kneser-Ney smoothing (Kneser and Ney, 1995; Chen and Goodman, 1998) and default pruning settings that remove singletons of order 3 and higher. Unlike Germann et al. (2009), the model was chosen to be small enough that all benchmarks would fit comfortably in physical memory. Benchmarks use the package's binary format; this work is also the fastest at building its binary format from the standard ARPA format. As noted in Section 4.4, disk cache state is controlled by reading the entire binary file before each test begins.

Time and memory consumption of each data structure is evaluated by computing perplexity on 4 billion tokens[9] from the Fourth English Gigaword corpus (Parker et al., 2009). Time is the sum of user and system CPU time consumed by the process minus time spent loading, converting strings to vocabulary identifiers, reading the input, or cleaning up. Time was then averaged over 4 billion queries. Resident memory usage was collected immediately after loading while peak virtual memory usage was collected immediately before termination. Table 4.2 shows results of the benchmark. Compared to decoding, this task is cache-unfriendly in that repeated queries happen only as they naturally occur in text. Therefore, performance is more closely tied to the underlying data structure than to the cache.

[8]This work was completed before Chapter 3.

[9]This figure includes the end of sentence marker </s> because it is queried after each sentence. It does not include the begin of sentence marker <s> because it is only used as context.

IRSTLM's cache made it slightly slower on this task, so results in Table 4.2 show IRSTLM without caching. The cache has a size parameter. Moses (Koehn et al., 2007) sets this parameter to 50, which resulted in a cache size of 2.82 GB. This is much larger than the cache size seen during translation (Section 4.5.3), perhaps because many of the queries were unique.

As suggested in the RandLM documentation, values were quantized to 8 bits and the false-positive probability was set to $\frac{1}{256}$.

Section 3.6.3 noted that BerkeleyLM requires one to guess how much memory it will use. The same protocol is followed here: binary search to find the lowest memory setting with which it would run, followed by empirical measurement of actual memory usage. Curiously, BerkeleyLM's memory usage peaked during loading and fell once loading completed; Pauls (2011) explained that loading entails additional data structures that are freed before querying.

The results in Table 4.2 show that the probing data structure is 81% faster than the trie, which is in turn 31% faster than the fastest baseline. Memory usage in the probing data structure is high, though SRILM is even larger, so where memory is of concern, the trie data structure should be used, possibly with the quantization and array compression options. Chapter 6 contributes another form of compression by collapsing probability and backoff. For even larger models, RandLM and ShefLM provide lossy compression options.

Package	Variant	Queries/ms	RAM (GB)
This Work	Probing	1818	5.28
	Trie	1139	2.72
	Trie 8 bits[a]	1127	1.59
	Array	436	2.33
	Array 8 bits[a]	461	1.19
SRI	Default	750	9.19
	Compact	238	7.27
IRST	Invert	426	2.91
	Default	368	2.91
	Invert 8 bits[a]	402	1.80
MIT	Default	410	7.72+1.34[b]
Rand	Backoff 8 bits[a]	56	1.30+2.82[b]
Berkeley	Hash+Scroll[a]	913	5.28+2.32[b]
	Hash[a]	767	3.71+1.72[b]
	Compressed[a]	126	1.73+0.71[b]
Estimates based on proportions			
Shef	C-MPHR[a]	607[c]	
TPT	Default	357[d]	

[a]Uses lossy compression.

[b]The first value reports resident memory use immediately after loading while the total is peak virtual memory usage. The difference is due to batch processing (MIT), caching (Rand), or extra data structures for loading (Berkeley).

[c]Based on the ratio to SRI's speed reported in Guthrie and Hepple (2010) under different conditions.

[d]Based on a comparison with SRI reported by Germann (2011) under different conditions.

Table 4.2: Single-threaded speed and memory usage to compute perplexity on 4 billion words. In most cases, resident memory after loading was close to peak virtual memory usage, so only one value is noted for these systems. Array refers to the trie data structure with array compression.

4.5.3 Translation

This task measures impact on performance of the Moses (Koehn et al., 2007) decoder. Only toolkits supported by Moses are included.[10] Table 4.3 shows both single-threaded results, mostly for comparison to IRSTLM, and multi-threaded results.

The phrase-based translation system was built accoding to the baseline instructions for the French–English track of the 2011 Workshop on Machine Translation (Callison-Burch et al., 2011). The language model is the same used in the previous section. The phrase table was extracted from Europarl (Koehn, 2005). The test set has 3003 sentences.

Feature weights were originally tuned for exact representations using minimum error rate training (Och, 2003). Because some representations are lossy, feature weights were also retuned. The retuned systems generally used slightly less resources.

[10]ShefLM is excluded because its Moses integration was never optimized. Chow (2012) reported that it took twice as long as with RandLM.

Unlike the previous section, the entire cost of the process is reported, including load time. However, the decoder was modified to terminate immediately instead of cleaning up, because SRILM's compact format would otherwise take hours to clean up. Binary language models and the text phrase table were forced into the disk cache before each run. These conditions make the values appropriate for estimating repeated run times, such as in parameter tuning.

Resident and peak virtual memory usage were collected immediately before termination. With memory mapping, resident usage reports only the parts of the language model that were used by the process (in atomic 4096-byte pages). Virtual memory usage includes the entire language model, even if parts were not loaded. The gap between resident and virtual memory is also due, in part, to Moses allocating memory that it did not completely use or allocating memory then freeing it later.

Overall, language modeling significantly impacts decoder performance. In line with performance on the perplexity task (Table 4.2), the probing model is the fastest followed by the trie, and subsequently other packages. The trie model continues to use the least memory of the non-lossy options. For RandLM and IRSTLM, the effect of caching can be seen on speed and memory usage. This is most severe with RandLM in the multi-threaded case, where each thread keeps a separate cache, exceeding the original model size. As noted for the perplexity task, the cache is not expected to grow substantially with model size, so RandLM remains a low-memory option. Caching for IRSTLM is smaller at 0.09 GB resident memory, though it supports only a single thread. The BerkeleyLM direct-mapped cache is in principle faster than caches implemented by RandLM and by IRSTLM.

One Thread

		Time (m)		RAM (GB)	
Package	**Variant**	**CPU**	**Wall**	**Res**	**Virt**
This Work	Probing-L	72.3	72.4	7.83	7.92
	Probing-P	73.6	74.7	7.83	7.92
	Trie-L	80.4	80.6	4.95	5.24
	Trie-P	80.1	80.1	4.95	5.24
	Trie-L 8[a]	79.5	79.5	3.97	4.10
	Trie-P 8[a]	79.9	79.9	3.97	4.10
SRI	Default	85.9	86.1	11.90	11.94
	Compact	155.5	155.7	9.98	10.02
IRST	Cache-Invert-L	106.4	106.5	5.36	5.84
	Cache-Invert-R	106.7	106.9	5.73	5.84
	Invert-L	117.2	117.3	5.27	5.67
	Invert-R	117.7	118.0	5.64	5.67
	Default-L	126.3	126.4	5.26	5.67
	Default-R	127.1	127.3	5.64	5.67
Rand	Backoff[a]	277.9	278.0	4.05	4.18
	Backoff[b]	247.6	247.8	4.06	4.18

Eight Threads

		Time (m)		RAM (GB)	
Package	**Variant**	**CPU**	**Wall**	**Res**	**Virt**
This Work	Probing-L	130.4	20.2	7.91	8.53
	Probing-P	132.6	21.7	7.91	8.41
	Trie-L	132.1	20.6	5.03	5.85
	Trie-P	132.2	20.5	5.02	5.84
	Trie-L 8[a]	137.1	21.2	4.03	4.60
	Trie-P 8[a]	134.6	20.8	4.03	4.72
SRI	Default	153.2	26.0	11.97	12.56
	Compact	243.3	36.9	10.05	10.55
Rand	Backoff[a]	346.8	49.4	5.41	6.78
	Backoff[b]	308.7	44.4	5.26	6.81

[a]Lossy compression with the same weights.
[b]Lossy compression with retuned weights.

Table 4.3: Cost to translate 3003 sentences with Moses. Where applicable, models were loaded with lazy memory mapping (-L), prefaulting (-P), and normal reading (-R). IRST is not threadsafe. Loading is single-threaded and some operations involve locking.

4.5.4 Comparison with RandLM

In addition to storing backoff-smoothed models (Katz, 1987), RandLM supports stupid backoff (Brants et al., 2007). With stupid backoff, only counts are stored, saving memory over probability and backoff (though Chapter 5 shows how to compress backoff-smoothed models). RandLM was used to build an unpruned stupid backoff model on the same data as in Section 4.5.2. Because the previous model was pruned, IRSTLM was used to estimate an unpruned approximate Kneser-Ney language model (Section 3.2.4) on the same data in three pieces. Table 4.4 shows the results. Moses was run single-threaded to minimize the impact of

			RAM (GB)		
Package	**Variant**	**Time (m)**	**Res**	**Virt**	**BLEU**
This Work	Trie	82.9	12.16	14.39	27.24
	Trie 8 bits	82.7	8.41	9.41	27.22
	Trie 4 bits	83.2	7.74	8.55	27.09
Rand	Stupid 8 bits	218.7	5.07	5.18	25.54
	Backoff 8 bits	337.4	7.17	7.28	25.45

Table 4.4: CPU time, memory usage, and uncased BLEU (Papineni et al., 2002) score for single-threaded Moses translating the same test set. Each lossy model was run twice: once with specially-tuned weights and once with weights tuned using an exact model. The difference in BLEU was minor and the better result is reported.

RandLM's cache on memory use. RandLM is the clear winner in RAM utilization, but is also slower and lower quality. However, the point of RandLM is to scale to even larger data, compensating for this loss in quality.

4.6 Summary

Efficient querying is crucial to almost all applications of language modeling. This chapter described two data structures for language modeling that achieve substantial reductions in time and memory cost. The probing data structure is 2.4 times as fast as SRILM and uses less memory too. The trie data structure uses less memory than the smallest lossless alternative and is still faster than all baseline toolkits. These performance gains transfer to improved system runtime performance; while experiments focused on phrase-based translation with Moses (Koehn et al., 2007), the implementation, dubbed KenLM, has been widely applied in machine translation (Dyer et al., 2010; Li et al., 2009; Vilar et al., 2010) and other areas (Kim et al., 2012; Si et al., 2013) . Several optimizations led to these results: hashing, custom lookup tables, and bit-level packing. Contributions in this chapter make subsequent work in Chapters 5, 6 and 7 more efficient.

Chapter 5

Left and Right State[1]

It is a wise father that knows his own child. –William Shakespeare, *The Merchant of Venice*

The last thing one knows in constructing a work is what to put first. –Blaise Pascal

From the language model's perspective, perplexity computation and decoding boil down to string concatenation. When strings are concatenated, the language model probability changes due to cross-boundary n-grams. *State* formalizes what the language model needs to know about each string in order to compute this change in probability. Decoders can recombine hypotheses if their states are equal. State also carries information from one query to the next, reducing the cost of each query. Chapter 6 changes how probabilities are estimated for words in state. Chapter 7 uses state as the basis for a new search algorithm.

5.1 Introduction

Language model probability changes when a decoder concatenates e.g. "San" and "Francisco" to form "San Francisco".[2] Concatenation can increase or decrease language model probability; in this example it increases.

$$p(\text{San Francisco}) = p(\text{San})p(\text{Francisco} \mid \text{San}) \gg p(\text{San})p(\text{Francisco})$$

Section 2.4.2 defined correction factor CORRECT that measures the effect.[3]

$$\textsc{Correct}(\text{San} \bullet \text{Francisco}) = \frac{p(\text{San Francisco})}{p(\text{San})p(\text{Francisco})} \gg 1$$

This chapter is about defining a state function s that distills and encodes everything the language model needs to know about a string in order to compute CORRECT. State is designed so that it can be used in lieu of a string.

$$\textsc{Correct}(s(\text{San}) \bullet s(\text{Francisco})) = \left(\frac{p(\text{San Francisco})}{p(\text{San})p(\text{Francisco})}, s(\text{San Francisco})\right)$$

Because state carries additional information, such as backoff weights, computing the correction factor is faster when the decoder passes state instead of strings (Section 5.4). Moreover, because CORRECT is a function, hypotheses with equal state can be recombined by a decoder.

[1]Earlier versions of this work appear in Heafield (2011) and Heafield et al. (2011).

[2]A translation system would likely handle "San Francisco" as a phrasal unit. The language model is also responsible for scoring words in phrases.

[3]This is also known as mutual information, though the term correction factor clarifies that its role is solely to assist in computing language model probability.

With an N-gram language model, CORRECT examines at most the N–1 rightmost words of the first string (its *right state*) and at most the N–1 leftmost words of the second string (its *left state*). In a 4–gram language model,

$$\text{CORRECT}(\text{<s> the}\underbrace{\boxed{\vdash \text{boy saw the}}}_{\text{Right State}}\bullet\underbrace{\boxed{\text{man with a} \dashv}}_{\text{Left State}}\text{telescope .}) = \text{CORRECT}(\underbrace{\boxed{\vdash \text{boy saw the}}}_{\text{Right State}}\bullet\underbrace{\boxed{\text{man with a} \dashv}}_{\text{Left State}})$$

The turnstiles $\dashv$ and $\vdash$ indicate that left and right state are complete, respectively. This distinction matters when a string is too short to have a complete state. For example,

$$\text{CORRECT}(\underbrace{\boxed{\rhd \text{ in San}}}_{\text{Right State}} \bullet \underbrace{\boxed{\text{Francisco} \lhd}}_{\text{Left State}}) \neq \text{CORRECT}(\underbrace{\boxed{\rhd \text{ San}}}_{\text{Right State}} \bullet \underbrace{\boxed{\text{Francisco} \lhd}}_{\text{Left State}})$$

because the probability of "Francisco" depends on "in". Short state is denoted with symbols $\lhd$ for left state and $\rhd$ for right state. In general strings can be concatenated on either side, so they have both left and right state. Following Li and Khudanpur (2008), left and right state are written as a pair delimited by $\diamond$ i.e. (San $\lhd \diamond \rhd$ San). Table 5.1 summarizes the notation introduced in this chapter. A longer example is shown in Figure 5.1.

Notation	**Definition**
$w_1^{\lvert w\rvert}$	A string.
ϵ	The empty string.
$s(w_1^{\lvert w\rvert})$	The state of string $w_1^{\lvert w\rvert}$.
$\text{CORRECT}(v_1^{\lvert v\rvert} \bullet w_1^{\lvert w\rvert})$	Change in probability $\frac{p(v_1^{\lvert v\rvert} w_1^{\lvert w\rvert})}{p(v_1^{\lvert v\rvert})p(w_1^{\lvert w\rvert})}$ when strings $v_1^{\lvert v\rvert}$ and $w_1^{\lvert w\rvert}$ are concatenated.
$\diamond$	Delimits a pair of left state and right state. Stands for elided words.
$\dashv$	Completes left state. Nothing beyond this marker matters when extending left.
$\vdash$	Completes right state. Nothing beyond this marker matters when extending right.
$(w_1^{\text{left}} \dashv \diamond \vdash w_{\text{right}}^{\lvert w\rvert})$	The state of string $w_1^{\lvert w\rvert}$ when it is long enough to have complete state.
$\lhd$	Denotes a short left state.
$\rhd$	Denotes a short right state.
$(w_1^{\lvert w\rvert} \lhd \diamond \rhd w_1^{\lvert w\rvert})$	The state of string $w_1^{\lvert w\rvert}$ when it is too short to have complete state.

Table 5.1: Notation for states.

$$\underbrace{\boxed{\text{countries that maintain}}}_{\text{Left State}} \text{ diplomatic relations with } \underbrace{\boxed{\text{North Korea .}}}_{\text{Right State}}$$

Figure 5.1: A string with state (countries that maintain $\dashv \diamond \vdash$ North Korea .). Section 5.3 shows how to encode fewer words in some cases.

When strings have equal state, the language model will treat them exactly the same going forward. Decoders use this to determine when it is safe to recombine hypotheses. More recombination means each hypothesis reasons over more derivations at once. Decoders can then search more derivations (increasing accuracy) or pack derivations into fewer hypotheses (decreasing CPU time). Section 5.3 is about the first goal in this chapter: encode as few words as possible in state, increasing the chance that states will be equal and ultimately improving the time-accuracy trade-off presented by approximate search.

The second goal (Section 5.4) is to increase query speed by carring useful information in state from one query to the next. Many tasks, such as perplexity computation, query the language model in a left-to-right scrolling pattern $p(w_N \mid w_1^{N-1})$, $p(w_{N+1} \mid w_2^N)$, $p(w_{N+2} \mid w_3^{N+1})$, etc. Each query can be made faster by carrying information, in this case backoff weights, from one query to the next. Another optimization applies when strings are extended to their left.

Some decoders build hypotheses from left to right i.e. phrase-based Moses (Koehn et al., 2007) or left-to-right syntactic decoders (Watanabe et al., 2006; Huang and Mi, 2010). In these cases, hypotheses have no need for left state because they will never be extended to their left. The right state optimizations in this chapter still apply.

5.2 The State Abstraction

This section describes how state is intended to be used by a decoder. The abstraction can be used instead of—or in addition to—direct queries. The benefit of this abstraction is that hides optimizations presented in the later sections and enables Chapter 6.

Generally, the decoder is expected to remember the state of each string it builds. This is not a requirement, but simply faster. For example, Moses (Koehn et al., 2007) does not store state with each target phrase because doing so would cost too much memory. Every time a string grows by appending a word or by concatenation, the language model provides the new state.

Strings are built inductively from left to right. The base cases are $s(\epsilon)$ for empty state or $s(\text{<s>})$ for the beginning of sentence context. The inductive case is the APPEND function.

$$\textsc{Append}(s(\text{San}) \bullet \text{Francisco}) = (p(\text{Francisco} \mid \text{San}), s(\text{San Francisco}))$$

This functionality is sufficient to build any string,[4] though the CORRECT function described in Section 5.1 is useful for bottom-up decoding. No matter what sequence of CORRECT and APPEND operations are used to build a string, the probabilites multiply to form the string's probability. For example, Algorithm 4 shows how perplexity is computed by building an entire sentence from left to right.

```
total ← 1
state ← $s(\text{<s>})$
for $i \leftarrow 1$ to $|w|$ do
    $(p(w_i \mid w_1^{i-1}), \text{state}) \leftarrow \textsc{Append}(\text{state}, w_i)$
    total ← total $\cdot\, p(w_i \mid w_1^{i-1})$
    // state = $s(\text{<s>}\ w_1^i)$
    // total = $p(w_1^i \mid \text{<s>})$
end
return total
```

Algorithm 4: Computing $p(w_1^{|w|} \mid \text{<s>})$ using state.

Using the functions CORRECT and APPEND has several advantages for decoders over normal queries:

- State encodes as few words as possible (Section 5.3), increasing the chances that states will be equal (and therefore recombinable).
- Queries are faster because state carries information from one query to the next (Section 5.4).
- Contributions in Chapter 6 are hidden behind CORRECT and APPEND, so decoders also gain support for those optimizations.

There are also disadvantages: the need to modify the decoder to handle state and extra memory consumption.

[4]The end of sentence symbol </s> simply acts as a normal word.

However, when implemented in Moses (Koehn et al., 2007), memory consumption actually decreased, albeit not substantially, because the preexisting representation was less efficient.

The Moses (Koehn et al., 2007; Hoang et al., 2009), cdec (Dyer et al., 2010), Joshua (Li et al., 2009), and Jane (Vilar et al., 2010) decoders have all been modified by their developers to use the state abstraction. Consequently, all of them support the optimizations in this chapter and in Chapter 6).

5.3 Minimizing State

As explained in Section 5.1, decoders perform better when state encodes fewer words. This section is about determining what words can safely be omitted from state and doing so efficiently.

State minimization uses a principle put forward by Li and Khudanpur (2008): if it can be proven that the language model will always back off (Section 2.2) before examining a word, then this word can be elided from the state. Specifically, if left state v_1^m has the property that xv_1^m does not appear in the language model for all words x, then v_m may be elided from the state. Similarly, if right state w_1^n has the property that $w_1^n x$ does not appear in the language model for all words x, then w_1 may be elided from the state. These statements apply recursively, so additional words such as v_{m-1} and w_2 may be similarly elided. Once all qualifying words have been elided, the state is said to be *minimized*. The criterion here is minimal in the sense that there exists some surrounding for which every surviving word matters.

Section 5.3.1 reviews existing work on state minimization. Section 5.3.2 shows how the criterion is computed efficiently. Finally, Section 5.3.3 shows that the criterion in the previous paragraph preserves correctness.

5.3.1 Related Work

Language model implementations vary in the extent to which they minimize state. As right state minimization was already commonly and efficiently implemented, it is the baseline for most experiments.

In addition to providing probability, IRSTLM (Federico et al., 2008) and BerkeleyLM (Pauls and Klein, 2011) indicate the length of the n-gram matched. This can be used as a state minimization heuristic. For example, if $p(v_n \mid v_1^{n-1})$ backed off to $p(v_n \mid v_2^{n-1})$, then v_n can be elided from left state. However, this heuristic misses some opportunities for state minimization. For example, “relations with North Korea” might appear in the language model while, due to pruning or filtering, “relations with North Korea x” does not appear in the language model for all words x. In this case, IRSTLM and BerkeleyLM will miss the opportunity to elide “relations” from right state.

SRILM (Stolcke, 2002) performs full right state minimization. However, probability and the length of right state are exposed by separate calls to the language model, each of which performs a full search through the data structure. There is no method to retrieve the length of the n-gram matched by a query, so the limited form of left state minimization possible with IRSTLM and BerkeleyLM is not possible with SRILM.

RandLM (Talbot and Osborne, 2007) supports full right state minimization. For left state, it is similar to IRSTLM and BerkeleyLM in that both indicate the n-gram length matched.

Prior to this work, Moses (Hoang et al., 2009) implemented right state minimization to the extent supported by toolkits. Left state was generally N–1 words, with the exception that hypotheses beginning with <s> had empty left state, a special case of state minimization.

Li and Khudanpur (2008) put forward the rules for state minimization used here. They fully minimized both left and right state, with the exception of strings shorter than N–1 words. This work goes further by also minimizing the state of strings shorter than N–1 words. For example, the unknown word <unk> usually has empty left state and empty right state.[5]

[5]Some language models replace low-count words with <unk>. In these models, it is possible to have n-grams containing

While Li and Khudanpur (2008) described an "inefficient implementation of the prefix- and suffix-lookup", here the necessary information is stored with each n-gram entry, incurring minimal overhead as described in Section 5.3.2. Indeed, their state minimization work was originally performed to improve performance of the Joshua decoder (Li et al., 2009). Joshua has since adopted this work (Post et al., 2013).

5.3.2 Efficient Implementation

In order to minimize state, the language model needs to determine whether a given n-gram might extend to the left or right. This information is precomputed for every n-gram and stored with its entry.

Left state extension information is stored in a data-structure dependent way. In the byte-aligned probing data structure from Chapter 4, the sign bit of the log probability would normally always be negative. This bit is repurposed to indicate whether the n-gram extends left. With the trie data structure described in Chapter 4, the record $w_1^{|w|}$ stores a pointer to the beginning of the block of $xw_1^{|w|}$ for all words x. The record that follows $w_1^{|w|}$ similarly has a pointer to the beginning of its leftward extensions. These pointers are equal if and only if there is no leftward extension.

Right extension information is stored as part of the backoff weight. If an n-gram does not extend to the right, then it generally has log backoff zero.[6] Conveniently, there are two ways to represent zero as an IEEE floating-point value on a machine: $+0.0$ and -0.0. The value $+0.0$ indicates n-grams that do extend to the right while -0.0 indicates n-grams that do not extend to the right.[7] When quantized (Whittaker and Raj, 2001), the model reserves two values for zero backoff to achieve the same effect.

As noted in Chapter 4, the language model retrieves $p(w_{|w|} \mid w_1^{|w|-1})$ by visiting $w_{|w|}$, $w_{|w|-1}^{|w|}$, $w_{|w|-2}^{|w|}$, etc. These are the same records whose values are needed to make a full determination of state minimization.

5.3.3 Correctness and Scoring

It remains to show that probability can be computed given only the information in state. The proof relies on the *substring property* of language models:

Property 1. *If the n-gram $w_1^{|w|}$ appears in a language model, then so do substrings w_i^j for all* $1 \leq i \leq j \leq |w|$.

For unpruned models, the substring property follows naturally from estimation: when $w_1^{|w|}$ occurs in the corpus, all of its substrings are also extracted. However, as mentioned in Section 4.2.2, models pruned by SRILM (Stolcke, 2002) violate this property. The issue is addressed by hallucinating missing n-grams until the substring property holds. The impact on size is mimimal: a 1.2% increase in the number of n-grams.

In addition to the substring property, state minimization exploits the backoff procedure described in Section 4.1. When queried with $p(w_{|w|} \mid w_1^{|w|-1})$, the model finds the longest match $w_f^{|w|}$ then evaluates

$$p(w_{|w|} \mid w_1^{|w|-1}) = p(w_{|w|} \mid w_f^{|w|-1}) \prod_{i=1}^{f-1} b(w_i^{|w|-1}) \tag{5.1}$$

where conditional probability p and backoffs b were estimated when the model was built.

<unk>. Normal state minimization rules apply in either case.

[6]This is true of models generated directly by Chapter 3, SRILM, or IRSTLM. Filtering (Heafield and Lavie, 2010) may have removed the rightward extension; in this case, the current scheme will miss a recombination opportunity. An alternative would be to fold the backoff into the probability since it will always be charged, a special case of the optimization described in Chapter 6, though this has other effects such as no longer encoding a probability distribution.

[7]The trie representation in Chapter 4 was designed to be general-purpose, so it still retains the sign bit for log backoff, even though this bit is always negative for models estimated in Chapter 3.

Left State

Proposition 2. *The* CORRECT *function does not depend on words elided from left state. It does, however, depend on whether a word was elided, hence the distinction between* $\lhd$ *and* $\dashv$ *in Table 5.1.*

Proof. Let $v_1^{|v|}$ and $w_1^{|w|}$ be strings, where w_1^n is the left state of $w_1^{|w|}$.

$$\begin{aligned}
\textsc{Correct}(v_1^{|v|} \bullet w_1^{|w|}) &= \frac{p(v_1^{|v|} w_1^{|w|})}{p(v_1^{|v|})p(w_1^{|w|})} \\
&= \frac{\left(\prod_{i=1}^{|v|} p(v_i \mid v_1^{i-1})\right)\left(\prod_{i=1}^{|w|} p(w_i \mid v_1^{|v|} w_1^{i-1})\right)}{p(v_1^{|v|})p(w_1^{|w|})} \\
&= \frac{p(v_1^{|v|})\left(\prod_{i=1}^{|w|} p(w_i \mid v_1^{|v|} w_1^{i-1})\right)}{p(v_1^{|v|})p(w_1^{|w|})} \\
&= \frac{\prod_{i=1}^{|w|} p(w_i \mid v_1^{|v|} w_1^{i-1})}{p(w_1^{|w|})} \\
&= \frac{\prod_{i=1}^{|w|} p(w_i \mid v_1^{|v|} w_1^{i-1})}{\prod_{i=1}^{|w|} p(w_i \mid w_1^{i-1})} \\
&= \prod_{i=1}^{|w|} \frac{p(w_i \mid v_1^{|v|} w_1^{i-1})}{p(w_i \mid w_1^{i-1})} \qquad (5.2)
\end{aligned}$$

For $i \leq n$, the terms depend only on v and left state w_1^n. If no word has been elided ($n = |w|$), then CORRECT can be computed immediately. Otherwise, it remains to show how to compute the terms of Equation 5.2 for $i > n$.

By the left state minimization criterion, $v_{|v|} w_1^{n+1}$ does not appear in the model. It follows from the substring property that $v_j^{|v|} w_1^i$ does not appear for all j and for all $i > n$. Therefore, the backoff procedure evaluates

$$p(w_i \mid v_1^{|v|} w_1^{i-1}) = p(w_i \mid w_1^{i-1}) \prod_{j=1}^{|v|} b(v_j^{|v|} w_1^{i-1}) \; \forall i > n$$

so the terms of Equation 5.2 are given by

$$\frac{p(w_i \mid v_1^{|v|} w_1^{i-1})}{p(w_i \mid w_1^{i-1})} = \prod_{j=1}^{|v|} b(v_j^{|v|} w_1^{i-1}) \; \forall i > n$$

There are two cases:

$i > n + 1$ The string $v_{|v|} w_1^{n+1}$ does not appear as noted previously. Because $i > n + 1$ and i is an integer, $i - 1 \geq n + 1$. Thus, $v_{|v|} w_1^{n+1}$ is a substring of $v_j^{|v|} w_1^{i-1}$ and it follows that $v_j^{|v|} w_1^{i-1}$ does not appear by the substring property. By the definition of backoff, $b(v_j^{|v|} w_1^{i-1}) = 1$ for all j and all $i > n + 1$. Therefore,

$$\frac{p(w_i \mid v_1^{|v|} w_1^{i-1})}{p(w_i \mid w_1^{i-1})} = 1$$

$i = n+1$ Substituting the value of i into $b(v_j^{|v|} w_1^{i-1})$ yields $b(v_j^{|v|} w_1^n)$. These do not depend on w_{n+1}.

$$\frac{p(w_{n+1} \mid v_1^{|v|} w_1^n)}{p(w_{n+1} \mid w_1^n)} = \prod_{j=1}^{|v|} b(v_j^{|v|} w_1^n)$$

Taking the above into account, Equation 5.2 expands as

$$\textsc{Correct}(v_1^{|v|} \bullet w_1^{|w|}) = \left(\prod_{i=1}^{n} \frac{p(w_i \mid v_1^{|v|} w_1^{i-1})}{p(w_i \mid w_1^{i-1})} \right) \begin{cases} 1 & \text{if no word was elided} \\ \prod_{j=1}^{|v|} b(v_j^{|v|} w_1^n) & \text{otherwise} \end{cases} \tag{5.3}$$

□

In order to evaluate Equation 5.3, it is necessary to distinguish cases where left state minimization applies. These cases are denoted with $\lhd$ and $\dashv$, respectively, and stored as a bit in the implementation. The bit introduces some risk that states will not be equal. Such cases can be reduced by redefining the bit to indicate whether left state is complete (namely, growing the string to the right will never change its left state). If a string has length N–1, then left state is always complete and Equation 5.3 will not change because the backoff is always 1 for strings $v_j^{|v|} w_1^n$ longer than N–1 words. A similar situation occurs when right state minimization was able to elide a word. If right state minimization elided w_1 from $w_1^{|w|}$ then, by the right state criterion, $w_1^{|w|} x$ does not appear for any word x. By the substring property, $v w_1^{|w|} x$ does not appear for all words v and x. Thus, it can be predicted that, were x to be appended, the left state minimization criterion would elide x from $w_1^w x$. The effect on Equation 5.3 is that backoff is charged before x is known.

Right State

Proposition 3. *The backoff procedure can be evaluated correctly after right state minimization.*

Proof. Let $w_1^{|w|}$ be a string whose right state minimizes to $w_n^{|w|}$. Due to the minimization criterion, $w_{n-1}^{|w|} x_1$ does not appear in the model for any word x_1. Applying the substring property, $w_i^{|w|} x_1^{|x|}$ does not appear for any $i < n$ and any string $x_1^{|x|}$. Therefore, when queried for $p(x_j \mid w_i^{|w|} x_1^{j-1})$ for any i and j, the backoff procedure will base probability on $p(x_j \mid w_f^{|w|} x_1^{j-1})$ where $f \geq n$ (or possibly even depend solely on x). Since p is a function and w_{n-1} is not an argument, the term $p(x_j \mid w_f^{|w|} x_1^{j-1})$ does not depend on w_{n-1}.

The backoff terms all have the form $b(w_i^{|w|} x_1^{j-1})$. The claim that $b(w_i^{|w|} x_1^{j-1}) = 1$ whenever $i < n$ is shown by cases.

$j > 1$ The minimization criterion requires that $w_{n-1}^{|w|} x_1$ does not appear, so $w_i^{|w|} x_1^j$ does not appear for $i < n$ by the substring property. Thus $b(w_{n-1}^{|w|} x_1^j) = 1$ for $i < n$ by definition.

$j = 1$ Right state minimization also requires that $b(w_{n-1}^{|w|}) = 1$ as noted in Section 5.3.2. Since the state minimization criterion is tested for each word before proceeding to the next, it must be the case that $b(w_i^{|w|}) = 1$ for all $i < n$.

Combining all these cases, the terms in Equation 5.1 either do not depend on w_{n-1} or are known to be 1. □

5.4 State Encoding

State could be encoded simply as an array of words along with the bit that flags short hypotheses. This section shows how to encode state in a way that makes queries faster. Left state and right state are encoded independently.

5.4.1 Related Work

When used with other language model toolkits, Moses (Koehn et al., 2007) encodes right state as an array of vocabulary identifiers and a pointer to the record found by the backoff procedure. The pointer is not used to speed language model queries, but rather as a unique identifier of an n-gram. It is faster to compare this single integer than it is to compare array of integers, so it is used as a sorting key.

SRILM (Stolcke, 2002) separately allocates every node in its trie data structure. The relative ordering of data structure pointers is therefore inconsistent across runs. This leads to slightly non-deterministic output when pointers are used to break ties. While this work also stores pointers in state, they are actually offsets into a large array, making their comparison deterministic.

After this work, BerkeleyLM (Pauls and Klein, 2011) encoded right state using a pointer into their trie. Unlike SRILM, their pointer can be used to continue left-to-right queries without knowing the corresponding vocabulary indices. Their state is therefore smaller in terms of memory usage. However, making use of this pointer requires that the trie have forward and backward pointers, increasing data structure memory consumption by 40%. While this work uses a larger state encoding, there is no need to store backward pointers in the trie. Because there are generally fewer hypotheses than n-grams, the memory cost of state is much lower in this work.

BerkeleyLM uses states to optimistically search for longer n-gram matches first and must perform twice as many random accesses to retrieve backoff information. The probing model described in Section 4.2 can perform optimistic searches by jumping to any n-gram without needing state and without any additional memory. However, this optimistic search would not visit the entries necessary for the optimization in Section 5.4.3. Though experiments do not directly compare state implementations, experiments in Section 4.5 indicate that the overall approach in this work is faster.

5.4.2 Left State

Normally, every language model query entails walking through a data structure to find the longest matching entry. Equation 5.3 contains several such queries.

$$\prod_{i=1}^{n} \frac{p(w_i \mid v_1^{|v|} w_1^{i-1})}{p(w_i \mid w_1^{i-1})}$$

The goal is to efficiently compute each probability in this equation.

As described in Chapter 7, the query $p(w_i \mid v_1^{|v|} w_1^{i-1})$ is normally executed by visiting nodes in increasing order: $w_i, w_{i-1}^i, \ldots, w_1^i, v_{|v|} w_1^i, \ldots, v_1^{|v|} w_1^i$. Many of these nodes were already visited when the query $p(w_i \mid w_1^{i-1})$ was executed earlier as part of scoring $w_1^{|w|}$. To avoid repeated visits to the same nodes, left state stores pointer(w_1^i), a pointer to entry w_1^i in the data structure, for each i. Remembering pointer(w_1^i) enables the language model to resume lookup at the node w_1^i and immediately retrieve $p(w_i \mid w_1^{i-1})$. From there, it can continue following the data structure to retrieve $p(w_i \mid v_1^{|v|} w_1^{i-1})$. Thus, the term

$$\frac{p(w_i \mid v_1^{|v|} w_1^{i-1})}{p(w_i \mid w_1^{i-1})}$$

is computed with far fewer lookups. An example is shown in Figure 5.2.

Left state w_1^n is encoded as

$$\left(\left\{\text{pointer}(w_1^i)\right\}_{i=1}^{n}, \text{bit}\right)$$

where the bit distinguishes when state is short because the string has too few words. Only pointer(w_1^n) is necessary because the other terms pointer(w_1), pointer$(w_1^2), \ldots,$ pointer(w_1^{n-1}) can be derived from it.

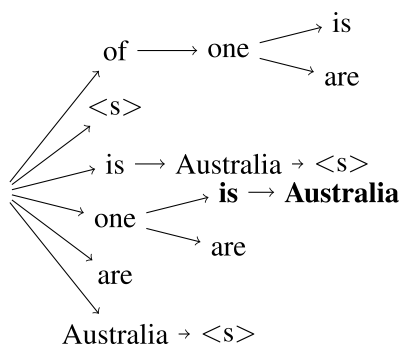


Figure 5.2: Extending left state "is one ⊣" to incorporate context "Australia" in a trie. By following the data structure pointer pointer(is one), lookup jumps to the node for "is one". The language model reads denominator $p\,(\text{one} \mid \text{is})$ then continues to "Australia is one" where it reads the numerator $p\,(\text{one} \mid \text{Australia is})$.

However, it is very inefficient to retrieve the other pointers, so they too are stored in state. For purposes of determining state equality, it suffices to compare pointer(w_1^n).

5.4.3 Right State

Normal query execution consists of retrieving the probability and any backoff penalties

$$p(x \mid w_{n+1}^{|w|}) = p(x \mid w_f^{|w|}) \prod_{i=n+1}^{f-1} b(w_i^n)$$

where $w_f^{|w|}x$ is the longest matching entry. The right state $w_{n+1}^{|w|}$ is encoded so that backoffs are immediately available instead of being retrieved from the data structure. Specifically, right state $w_{n+1}^{|w|}$ is encoded as

$$\left(w_{n+1}^{|w|}, \left\{b(w_i^{|w|})\right\}_{i=n+1}^{|w|}\right)$$

in anticipation that any of these backoff terms may be used.

When the language model was queried for $p(w_{|w|} \mid w_{n+1}^{|w|-1})$, it visited the nodes $w_{|w|}, w_{|w|-1}^{|w|}, \ldots, w_{n+1}^{|w|}$ in order to retrieve $p(w_{|w|} \mid w_{n+1}^{|w|-1})$. While visiting these nodes, the values $b(w_{|w|}), b(w_{|w|-1}^{|w|}), \ldots b(w_{n+1}^{|w|})$ were recorded to right state. The state carries this information to the next query $p(x \mid w_{n+1}^{|w|})$.

At first it seems that the two state encodings are incompatible. Specifically, left state enables a shortcut that skips nodes while right state uses those visits to retrieve backoff values. The issue arises when $v_1^{|v|}$ is concatenated with $w_1^{|w|}$ and there is a need to encode the right state of $v_1^{|v|}w_1^{|w|}$. The pointer in left state enables a shortcut to w_1^m where m was determined by left state minimization. Lookups continue with $v_{|v|}w_1^m$, $v_{|v|-1}^{|v|}w_1^m$, etc. These lookups retrieve $b(v_{|v|}w_1^m)$, $b(v_{|v|-1}^{|v|}w_1^m)$, etc. The remaining backoffs $\{b(w_i^m)\}_{i=1}^m$ were already encoded in the right state. If right state minimization elided a word from $w_1^{|w|}$, then the right state of $v_1^{|v|}w_1^{|w|}$ is simply copied.

5.5 Formal Definition of State

To summarize Sections 5.3 and 5.4, the formal definition of state is

$$s(w_1^{|w|}) = \left(\left\{\text{pointer}(w_1^i)\right\}_{i=1}^{\text{left}(w_1^{|w|})}, \text{bit}(w_1^{|w|}), w_{\text{right}(w_1^{|w|})}^{|w|}, \left\{b(w_i^{|w|})\right\}_{i=\text{right}(w_1^{|w|})}^{|w|} \right)$$

where the left state minimization criterion is

$$\text{left}(w_1^{|w|}) = \begin{cases} |w| & \text{if } |w| = 0 \text{ or } xw_1^{|w|} \text{ does not appear for all words } x \\ \text{left}(w_1^{|w|-1}) & \text{otherwise} \end{cases}$$

the right state minimization criterion is

$$\text{right}(w_1^{|w|}) = \begin{cases} 1 & \text{if } |w| = 0 \\ 1 & \text{if } w_1^{|w|}x \text{ does not appear for all words } x \text{ and } b(w_1^{|w|}) = 1 \\ 1 + \text{right}(w_2^{|w|}) & \text{otherwise} \end{cases}$$

and the bit indicates if state was short due to minimization ($\dashv$) or because the string had few words ($\lhd$)

$$\text{bit}(w_1^{|w|}) = \begin{cases} \dashv & \text{if } |w| \geq N-1, \text{left}(w_1^{|w|}) \neq |w|, \text{ or right}(w_1^{|w|}) \neq 1 \\ \lhd & \text{otherwise} \end{cases}$$

5.6 Experiments

Experiments measure the impact of state minimization and encoding on the Moses chart (Hoang et al., 2009) decoder. As right state minimization was already implemented in several other toolkits, the focus of experiments is on left state.

5.6.1 Systems

The experiments refer to five systems covering three language pairs built by Koehn (2011):

Chinese–English A hierarchical system trained on 2.1 million parallel news sentences. The language model is an interpolation of models built on the Xinhua and AFP portions of English Gigaword version 4 (Parker et al., 2009) plus the English side of the parallel data. The test set is 1357 sentences from the NIST 2008 evaluation. The uncased BLEU (Papineni et al., 2002) score is 29.63%.

German–English Two systems, one hierarchical and one with target-side syntax. Rules were extracted from Europarl (Koehn, 2005). The language model interpolates English Europarl, news commentary, and monolingual news released for the Workshop on Machine Translation (Callison-Burch et al., 2011). The official 3003-sentence test set is used. On this test set, the hierarchical and target syntax systems score 21.10% and 20.14% BLEU, respectively.

English–German A hierarchical system and a target-syntax system, both trained on Europarl. The language model consists of the German monolingual data released for the 2011 Workshop on Machine Translation: Europarl, news commentary, and monolingual news data from each year. As in the evaluation, the test set is the same 3003 sentences used for German-English but translated in the opposite direction. The uncased BLEU scores on this test set are 14.95% for hierarchical and 14.69% for target syntax.

These systems were designed for the University of Edinburgh's participation in the NIST Open MT Evaluation (Chinese–English) and Workshop on Machine Translation (German–English and English–German) using constrained data and the standard Moses pipeline. For target syntax models, the Collins parser (Collins, 1999) was used to parse the target side. Language models were built with order N=5 using SRILM[8] (Stolcke, 2002), interpolated modified Kneser-Ney smoothing (Kneser and Ney, 1995; Chen and Goodman, 1998), and the default pruning[9] that drops singletons of order three or above.

5.6.2 Recombination

The effectiveness of state minimization is measured in two ways: the extent to which it minimizes state length and the number of recombinations that happen as a result.

Table 5.2 shows the distribution of hypotheses by state length in the Chinese–English system. Left state minimization is generally more aggressive: 19.2% of left states are full length (4) compared to 29.5% of right states. The trend continues for shorter lengths as well. One reason for this difference is that the decoder applies the glue rule from left to right, generating hypotheses bound to the beginning of sentence. Hypotheses bound to the beginning of sentence cannot extend further left, and so their left state length is zero. The Moses baseline (Hoang et al., 2009) already implemented special-case handling for the beginning of sentence that allows hypotheses to recombine even if their first N–1 words differ, effectively dropping left state to zero length as well.

		Right Length					
		0	**1**	**2**	**3**	**4**	**Sum**
Left Length	**0**	0.3	1.1	3.0	3.2	3.2	10.7
	1	0.3	1.7	3.8	3.5	3.5	12.9
	2	0.6	2.9	9.5	8.8	8.5	30.3
	3	0.4	2.1	6.9	7.7	7.5	24.7
	4	0.3	1.6	5.1	5.6	6.7	19.2
Sum		1.9	9.4	28.3	28.8	29.5	97.8

Table 5.2: Percentage of hypotheses by length of left state and length of right state in the Chinese–English task with a cube pruning pop limit of 1000. On hypotheses considered by the decoder, left state minimization is more aggressive than right state minimization. Excluded from the table, 2.2% of hypotheses were shorter than 4 words, consisting of 0.1% unigrams, 1.1% bigrams, and 1.1% trigrams. Sums were computed prior to rounding.

To measure recombination, the decoder was run with left and right state minimization enabled. Each hypothesis keeps track of its highest-scoring surface form. When two hypotheses recombined, their highest-scoring surface forms were compared to determine if recombination would have been permitted without state minimization. In the Chinese–English system with a cube pruning pop limit of 1000, there were an average of 176,191 recombinations per sentence. Of these, 101,409 (57%) happened without any state minimization or were covered by the decoder's special handling of the beginning and end of sentence tokens. Left state minimization alone permitted 33,537 (19%) recombinations. Right state minimization alone permitted 26,111 (15%). Finally, 15,133 (9%) recombinations required both left and right state minimization.

Left state minimization is more aggressive and leads to more recombinations than does right state minimization.

[8]Chapter 3 was not implemented in 2011.

[9]Without pruning, the primary result that left state minimization changes the model score less than right state minimization still held.

5.6.3 Behavior Under Concatenation

The CORRECT function behaves differently, on average, for different state lengths. To measure this effect, the decoder was modified to record $\text{CORRECT}(v_1^{|v|}, w_1^{|w|})$ on every call, along with the length of right state in $s(v_1^{|v|})$, the length of left state in $s(w_1^{|w|})$, and the bit in the left state of $w_1^{|w|}$. The geometric average, or arithmetic average in log space, was then computed for each combination of lengths and the bit. Results are shown in Tables 5.3 and 5.4.

For long hypotheses (denoted by $\dashv$):

		Right Length			
		1	**2**	**3**	**4**
Left Length	**0**	−0.741	−1.062	−1.357	−1.701
	1	−0.269	−0.429	−0.588	−0.836
	2	−0.129	−0.236	−0.362	−0.567
	3	0.007	−0.061	−0.128	−0.314
	4	0.220	0.202	0.169	0.037

For short hypotheses (denoted by $\lhd$):

		Right Length			
		1	**2**	**3**	**4**
Left Length	**1**	0.017	−0.068	−0.174	−0.359
	2	0.096	0.046	−0.045	−0.239
	3	0.159	0.130	0.061	−0.117

Table 5.3: Mean $\log_{10}$ CORRECT on the Chinese-English system with pop limit 1000, reported separately by state lengths and by the left state flag. Zero length right state is not shown in the table because estimates were already based on empty context so the score adjustment is zero. Small hypotheses can only have lengths 1 through 3. Left state minimization predicts that scores will decline during concatenation while right state minimization predicts that scores will increase.

For large hypotheses (denoted by $\dashv$):

		Right Length			
		1	**2**	**3**	**4**
Left Length	**0**	−0.780	−1.074	−1.347	−1.573
	1	−0.181	−0.306	−0.455	−0.569
	2	0.023	−0.028	−0.100	−0.213
	3	0.184	0.199	0.190	0.111
	4	0.390	0.481	0.509	0.409

Table 5.4: Mean $\log_{10}$ score adjustments on the German-English hierarchical system with pop limit 1000. Entries show the same general trend as in Table 5.3, but the magnitude is larger and sometimes long right states fare better than do short right states.

Left state minimization predicts that scores will decline while right state minimization predicts that scores will increase. The difference lies in accounting for backoff weights. If left state is short, then it is known that the language model will back off, but it is not known what the backoff penalites will be, as they are context-dependent. Thus, charging the backoff penalties is delayed until CORRECT is called. If right state is short, the language model has backed off and backoff penalties have already been assessed because context was known. Moreover, there is less potential for future backoff, since there is less context to match.

These results suggest a simple scheme to improve search accuracy: use Table 5.3 as rest costs. Specifically, the CORRECT function is modified to lookup the appropriate value in Table 5.3 for every string it encounters. The value is added to the score of a string when it is generated and subtracted when the string is consumed. More formally,

$$\textsc{CorrectRest}(s(v_1^{|v|}), s(w_1^{|w|})) = \textsc{Correct}(s(v_1^{|v|}), s(w_1^{|w|})) + \textsc{Rest}(v_1^{|v|} w_1^{|w|}) - \textsc{Rest}(v_1^{|v|}) - \textsc{Rest}(w_1^{|w|})$$

where the REST function looks up the appropriate value in Table 5.3. These rest costs do improve search to some extent, but were abandoned because Chapter 6 works better.

5.6.4 Language Model Lookups

Speed improvements from left state encoding were measured by counting lookups and by timing the decoder. To translate 1357 sentences with a pop limit of 1000, the baseline Chinese-English system with right state minimization made a total of 760,788,076 pops. Turning on left state minimization led to a slight increase in pops at 761,124,349. Each pop entails evaluating the language model probability of the target side of a rule application, consisting of a string of terminals and non-terminals. To do so, it executes queries: one query for each terminal and one query for each word in the left state of a non-terminal. Left state minimization led to a 30% reduction in queries. The remaining queries perform data structure lookups, typically starting with unigrams and proceeding to larger n-grams. Storing pointers in left state allows the model to skip over lower-order n-grams. Table 5.5 shows the combined effect of reduced queries and data structure pointers.

n	**Baseline**	**Pointer**	**Reduction**
1	3,709,029,243	1,988,280,862	46%
2	3,305,400,924	2,128,356,570	35%
3	2,425,086,263	1,692,042,948	30%
4	998,098,720	752,425,908	24%
5	229,849,561	213,076,869	7%

Table 5.5: Number of language model data structure lookups for each n-gram length made by the Chinese-English system to translate 1357 sentences. Storing a data structure pointer in left state means queries can skip over shorter orders.

There is a substantial reduction in lookups. However, the left state pointers only skip over n-grams that have recently been retrieved and are likely to be cached by hardware. All five systems were run with pop limits 50, 100, 200, 500, and 1000. CPU time was reduced by only 3-6%. The disparity between reduction in lookups and overall CPU time reduction suggests that raw language model lookups are not a credible proxy for the computational cost of decoding. This is why experiments focus on CPU time as the measure of cost, despite its dependence on implementation.

5.6.5 Accuracy

Generally,[10] accuracy should improve because state minimization permits more recombination. Accuracy is measured by the model score of the single-best hypothesis found by search. Model scores have arbitrary scale and include constants, so only relative comparisons are meaningful. The model scores are averaged over all sentences in the test set. Table 5.6 shows the results for each system.

Both left state minimization and right state minimization improve model score. However, left state minimization alone is consistently worse than right state minimization alone. This seems to contradict the observation in Section 5.6.2 that left state minimization is more aggressive and responsible for more

[10] An increase in model score is not strictly guaranteed: a hypothesis that would have been pruned without recombination might instead be expanded, grow into additional hypotheses, and cause a component of a better hypothesis to be pruned.

	Hierarchical			Syntax	
	zh-en	**de-en**	**en-de**	**de-en**	**en-de**
None	−82.5435	−101.392	−79.2035	−104.244	−13.1425
Left	−82.5417	−101.390	−79.2028	−104.244	−13.1415
Right	−82.5368	−101.388	−79.1982	−104.242	−13.1389
Both	−82.5340	−101.387	−79.1972	−104.241	−13.1370

Table 5.6: Impact of state minimization on average model score. A higher model score is better. Model scores are scale-invariant and include a constant that indicates translation difficulty. All experiments used a cube pruning pop limit of 1000.

recombination. However, Section 5.6.3 provides a plausible explanation: when left state minimization applies, it predicts that hypotheses will fare poorly, so the recombined hypotheses are likely to be pruned.

5.6.6 Time-Accuracy

The model scores in Section 5.6.5 are difficult to interpret in isolation. By reducing the pop limit until average model score equals that of the baseline, search improvements can be interpreted as speed improvements. Figure 5.3 shows the trade-off between CPU time and average model score as moderated by the pop limit on the Chinese-English task. Timing measurements were performed on a 32-core machine with 64 GB of RAM. The language model and grammar were converted into binary format in advance then faulted into the operating system disk cache before each run.

With a pop limit of 1000, the baseline decoder yields average model score -82.5368 using 19.6 CPU seconds per sentence. Reducing the pop limit to 910 and applying left state minimization, the decoder yields average model score -82.5367 (better than the baseline) using 17.3 CPU seconds per sentence. This is a net 11% reduction in CPU time. Since accuracy has been traded for speed, there is no reason to expect a translation quality improvement. Using uncased BLEU (Papineni et al., 2002) as a proxy for translation quality, the baseline with pop limit 1000 scored 29.63% while left state minimization with pop limit 910 scored 29.67%, insignificantly better.

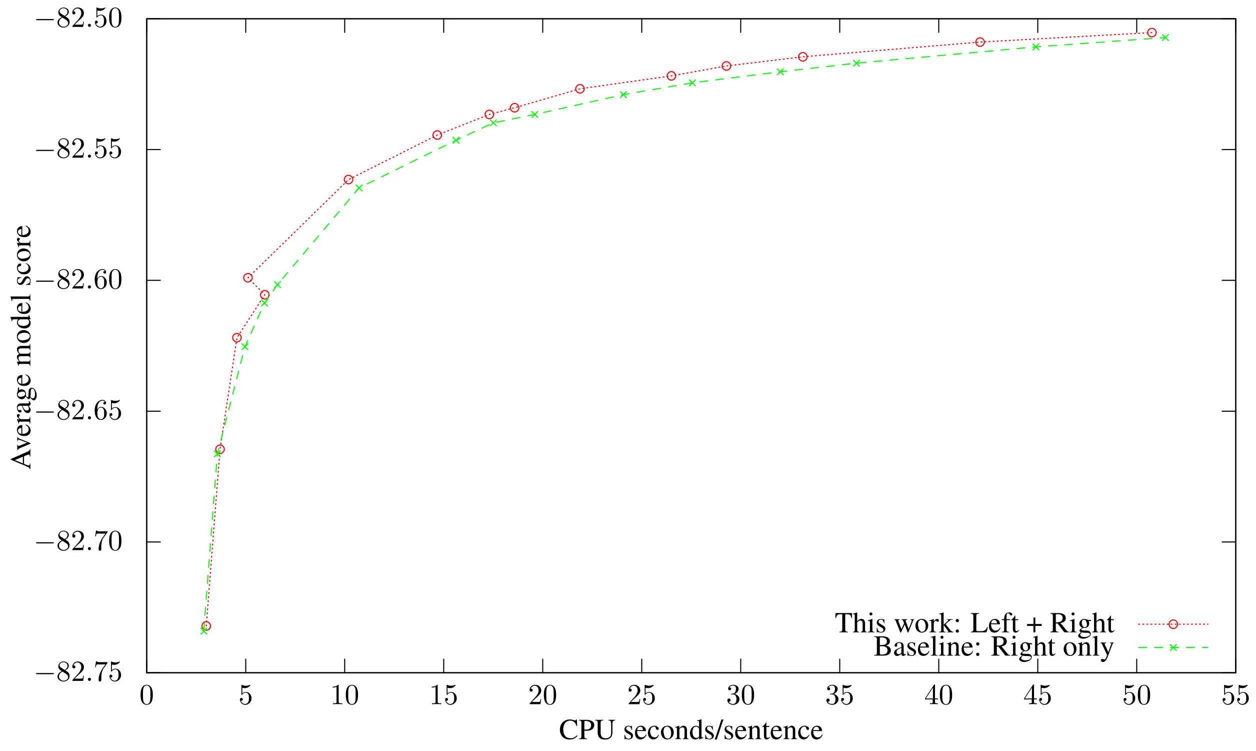
5.6.7 Memory

In terms of memory consumption, Moses's preexisting method for tracking state is less efficient than this work's encoding, so there was a small (about 1%) reduction in memory usage across the test systems using the same pop limit. There was no increase in language model storage size as explained in Section 5.3.2.

5.7 Summary

This chapter formalized language model state. Chapters 6 and 7 apply this formalism to save memory or improve search.

Two types of optimizations were presented: storing fewer words in state and carrying useful information from one query to the next. Compared with existing right state minimization, the improvement is relatively small (11% CPU time reduction). Most surprising is that left state minimization led to more recombination, but the improvement in quality is smaller than with right state. This was explained by the tendency of hypotheses with short left state to fare poorly in future concatenations, an observation that inspired Chapter 6.



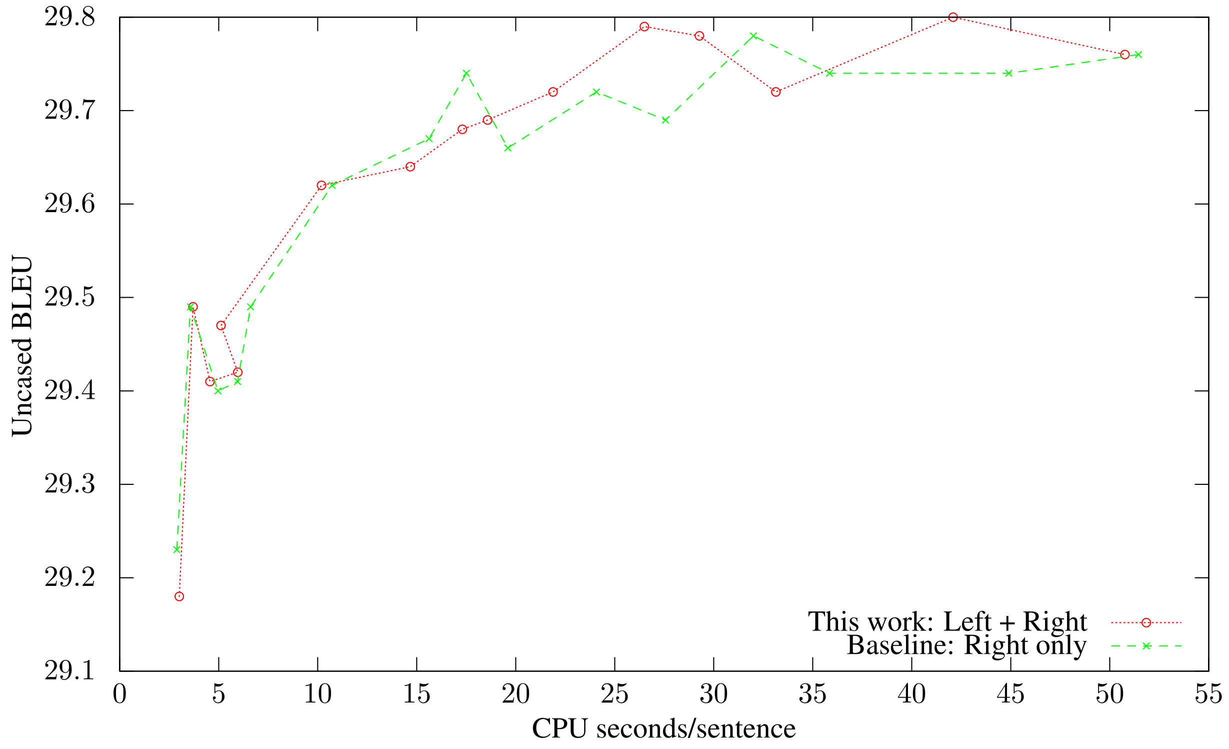


Figure 5.3: CPU seconds per sentence (user plus system), average model score, and BLEU on the Chinese-English task. Two curves are shown: full left and right state optimization and the commonly-implemented right state minimization.

Chapter 6

Rest Costs[1]

So so is good, very good, very excellent good: and yet it is not; it is but so so.
–William Shakespeare, *As You Like It*

There is more to life than simply increasing its speed. –Mahatma Gandhi

This chapter is about changing the scores of strings while preserving sentence-level probabilities. Such changes are called *rest costs* because they apply when strings are held by the decoder (and are, in some sense, at rest) but cancel out as strings are concatenated. Approximate search algorithms such as beam search, cube pruning (Chiang, 2007), and Chapter 7 base their pruning decisions on scores that include rest costs, so these changes impact search accuracy.

The first contribution improves search accuracy when Kneser-Ney language models (Kneser and Ney, 1995) are used, though at the expense of additional memory consumption. The second contribution collapses probability and backoff into a single value for any backoff-smoothed model (Katz, 1987), including Kneser-Ney, thereby saving memory but degrading search. The two contributions can also be stacked, using the same memory and slightly improving search.

6.1 Introduction

When the language model is called upon to score a string in isolation, the first few words have incomplete context and the last few words have not been completely used as context. Chapter 5 formalized these words as left and right state, respectively. The question in this chapter is how to score the words in state. The baseline is common practice (Chiang, 2007; Koehn et al., 2007; Dyer et al., 2010; Li et al., 2009; Vilar et al., 2010): simply provide the language model less context when it is indeterminate. Formally, the baseline score of string $w_1^{|w|}$ is given by

$$\prod_{i=1}^{|w|} p_N(w_i \mid w_1^{i-1})$$

where each w_i is a word and p_N is an N-gram language model. Subscripts are used to emphasize the order of the language model. When a subscript is not present (p), it refers to the full model (p_N).

[1]A prior version of this work was published as Heafield et al. (2012).

6.1.1 Better Rest Costs

The problem with the baseline estimate is that Kneser-Ney smoothing[2] (Kneser and Ney, 1995) assumes that the model has backed off when provided with fewer than N–1 words of context. Specifically, adjusted counts are specialized for lower-order entries as explained in Section 3.3.2. Conditional probabilities based on these adjusted counts assume that the language model has backed off. In this case, however, context is indeterminate, so it is inappropriate to use the lower-order probabilities to score the words in left state of a string.

As an example, 5–gram and unigram language models were built with Kneser-Ney smoothing on the same data. Strings frequently begin with "the". Using a lower-order entry from the 5–gram model,

$$\log_{10} p_5(\text{the}) = -2.49417$$

The unigram model does not condition on backing off, assigning

$$\log_{10} p_1(\text{the}) = -1.28504$$

Intuitively, the 5–gram model is surprised, by more than an order of magnitude, to see "the" without matching words that precede it.

This work remedies the situation by training N language models on the same data, one for each order. Each model p_n is an n-gram model (it has order n). Then p_n is used to score the nth word of a string. Thus, a unigram model scores the first word of a string, a bigram model scores the second word, and so on until left state is completely scored. Storing probabilities from these models requires one additional value per n-gram in the model, except for N-grams where this probability is already stored.

6.1.2 Saving Memory

Normally (Chapter 4), a backoff-smoothed model (Katz, 1987) is represented in memory by storing a probability and backoff with every n-gram (except full-length N-grams always have backoff one, so there is no need to store the backoff with each N-gram). This work proves that probability and backoff can be collapsed into a single value per n-gram while still retaining sentence-level probabilities.

Once probability and backoff are collapsed, non-sentence strings no longer have proper probabilities, but rather scores that approximate their probability. Experiments with cube pruning (Chiang, 2007) show this change has a negative impact on search accuracy, though the same level of accuracy can be obtained by raising the pop limit and spending more CPU time.

6.2 Related Work

Zens and Ney (2008) present rest costs for phrase-based translation. These rest costs are based on factors external to the hypothesis, namely output that the decoder may generate in the future. The rest costs in this work examine words internal to the sentence string, namely those in state. Theoretically, both could be used simultaneously. This work also differs by focusing on syntactic translation.

Carter et al. (2012) simultaneously published upper bounds on language model probability then applied them to exactly decode hidden Markov models. More recent work (Aziz et al., 2013) extended their approach to exact, albeit quite slow, decoding for hierarchical machine translation. Their upper bound implementation stores two additional values with each lower-order n-gram, an upper bound on probability and an upper bound on backoff.

[2]Other smoothing techniques, including Witten-Bell (Witten and Bell, 1991) and Good-Turing (Good, 1953), do not make this assumption.

Storing an upper bound on backoff is usually unnecessary. The bound must be at least one (in probability space) because the model might not back off and would therefore not charge any penalty. Section 3.4 proved that backoff is at most one in unpruned interpolated modified Kneser-Ney language models. Thus, the bound will always be one for such models and there is no need to store a constant with every entry in the language model. However, general models do allow the backoff term to be greater than one on rare occasions. In such cases, the backoff could be folded into probability, a limited version of the memory-saving optimization presented in this work.

Stupid backoff (Brants et al., 2007) is implemented by storing a count with each n-gram in memory. This has the advantage that there is only one value per n-gram instead of probability and backoff. However, computing a score on the fly from counts is only feasible with simple smoothing methods. This work shows that general backoff-smoothed language models (Katz, 1987) can stored using a single value per n-gram. The values are scores so minimal computation is performed at query time.

General related work on compressing language models is presented in Sections 2.3 and 4.3.

6.3 Contributions

6.3.1 Better Rest Costs

As mentioned in the introduction, the first few words of a string are typically scored using lower-order entries from an N-gram language model. However, Kneser-Ney smoothing (Kneser and Ney, 1995) conditions lower-order probabilities on backing off. This exploits the backoff algorithm, which bases its probability on the longest possible match. If an N-gram was seen in the training data, the model will match it fully and use the smoothed count. Otherwise, the full N-gram was not seen in the training data and the model resorts to a shorter n-gram match. Probability of this shorter match is based on how often the n-gram is seen in different contexts (the adjusted count), not how often it is seen in general (the raw count). Thus, probabilities associated with shorter n-grams are not representative of cases where context is short simply because additional context is indeterminate at the time of scoring.

For words elided by left state minimization from Section 5.3, the backoff assumption is correct because it has been proven that the language model will back off. For words that remain in left state, it is unknown if the model will back off. The idea is to use a language model of the same order to produce a rest cost. Specifically, there are N language models, one of each order from 1 to N. The models are trained on the same corpus with the same smoothing parameters to the extent that they apply. These are then compiled into one data structure where each n-gram record has three values:

1. Probability p_n from the n-gram language model if $n < N$
2. Probability p_N from the N-gram language model
3. Backoff b from the N-gram language model if $n < N$

For full-length N-grams, the two probabilities are equal and backoff is one, so only one value is stored. A small issue arises if the language models are pruned, in which case the n-gram model may not have a probability for every n-gram present in the N-gram model. These probabilities are constructed by querying the n-gram model in the normal way (Chapter 4). The idea is that p_n is the average conditional probability that will be encountered once additional context becomes known.

Formalizing the above, let $w_1^{|w|}$ be a string whose left state minimizes to w_1^m. The baseline score estimate is

$$p_{\text{baseline}}(w_1^{|w|}) = \left(\prod_{n=1}^{m} p_{\boldsymbol{N}}(w_n \mid w_1^{n-1})\right) \cdot \left(\prod_{n=m+1}^{N+1} p_N(w_n \mid w_1^{n-1})\right) \cdot \left(\prod_{n=N}^{|w|} p_N(w_n \mid w_{n-N+1}^{n-1})\right) \tag{6.1}$$

while the improved estimate is

$$p_{\text{improved}}(w_1^{m}) = \left(\prod_{n=1}^{m} p_{\boldsymbol{n}}(w_n \mid w_1^{n-1})\right) \cdot \left(\prod_{n=m+1}^{N+1} p_N(w_n \mid w_1^{n-1})\right) \cdot \left(\prod_{n=N}^{|w|} p_N(w_n \mid w_{n-N+1}^{n-1})\right) \tag{6.2}$$

The difference between these equations is that p_n is used for words in the left state i.e. $1 \le n \le m$. It is not necessary to store backoffs for p_n because m was chosen by left state minimization such that all queried n-grams appear in the model.

More involved estimates that interpolate the upper bound, the lower bound, p_n, and p_N did not improve results in any consistent way. The interpolation weights were tuned on cube pruning logs.

This modification to the language model improves rest costs (and therefore quality or CPU time) at the expense of using more memory to store p_n. The next section does the opposite: make rest costs worse to reduce storage size.

6.3.2 Saving Memory

Several smoothing strategies, including Kneser-Ney, are based on the backoff algorithm (Katz, 1987) shown in Algorithm 5. This is typically implemented by storing probability p and backoff b with each entry, as was done in Chapter 4. Given a query $w_1^{|w|}$, the backoff algorithm bases probability $p(w_{|w|} \mid w_1^{|w|-1})$ on as much context as possible. Formally, it finds the minimum f so that $w_f^{|w|}$ is in the model then uses $p(w_{|w|} \mid w_f^{|w|-1})$ as the basis for probability. Backoff penalties b are charged because a longer match was not found, forming the product

$$p(w_{|w|} \mid w_1^{|w|-1}) = p(w_{|w|} \mid w_f^{|w|-1}) \prod_{j=1}^{f-1} b(w_j^{|w|-1}) \tag{6.3}$$

Notably, the backoff penalties $\{b(w_j^{|w|-1})\}_{j=1}^{|w|-1}$ are independent of $w_{|w|}$, though which backoff penalties are charged depends on f.

To save memory, this contribution accounts for backoff in a different way, defining q

$$q(w_{|w|} \mid w_1^{|w|-1}) = \frac{p(w_{|w|} \mid w_f^{|w|-1}) \prod_{j=f}^{|w|} b(w_j^{|w|})}{\prod_{j=f}^{|w|-1} b(w_j^{|w|-1})}$$

where again $w_f^{|w|}$ is the longest matching entry in the model. The idea is that q is a term in the telescoping series that scores a string, shown in equation (6.1) or (6.2). The numerator pessimistically charges all backoff penalties, as if the next word $w_{|w|+1}$ will only match a unigram. When $w_{|w|+1}$ is scored, the denominator of $q(w_{|w|+1} \mid w_1^{|w|})$ cancels out backoff terms that were wrongly charged. Once these terms are canceled, all that is left is p, the correct backoff penalties, and terms on the edge of the series.

Proposition 4. *The terms of q telescope. Formally, let $w_1^{|w|}$ be a string and f take the minimum value so that $w_f^{|w|}$ is in the model. Then*

$$q(w_1^{|w|}) = p(w_1^{|w|}) \prod_{j=f}^{|w|} b(w_j^{|w|})$$

```
backoff ← $1$
for $f \leftarrow 1$ to $|w|$ do
    if $w_f^{|w|}$ is in the model then
        return $p(w_{|w|} \mid w_f^{|w|-1}) \cdot$ backoff
    else
        if $w_f^{|w|-1}$ is in the model then
            backoff ← backoff $\cdot\, b(w_f^{|w|-1})$
        end
    end
end
```

Algorithm 5: The baseline backoff algorithm to compute $p(w_{|w|} \mid w_1^{|w|-1})$. It always terminates with a probability because even unknown words are treated as a unigram.

```
for $f \leftarrow 1$ to $|w|$ do
    if $w_f^{|w|}$ is in the model then
        return $q(w_{|w|} \mid w_f^{|w|-1})$
    end
end
```

Algorithm 6: The run-time pessimistic backoff algorithm to find $q(w_{|w|} \mid w_1^{|w|-1})$. It assumes that q has been computed at model building time.

Proof. By induction on $|w|$. When $|w| = 1$, $f = 1$ since the word w_1 is either in the vocabulary or mapped to <unk> and treated like a unigram.

$$q(w_1) = \frac{p(w_1) \prod_{j=1}^{1} b(w_j^1)}{\prod_{j=1}^{0} b(w_j^0)} = p(w_1)b(w_1)$$

For $|w| > 1$,

$$\begin{aligned} q(w_1^{|w|}) &= q(w_1^{|w|-1}) q(w_{|w|} \mid w_1^{|w|-1}) \\ &= \frac{q(w_1^{|w|-1}) p(w_{|w|} \mid w_f^{|w|-1}) \prod_{j=f}^{|w|} b(w_j^{|w|})}{\prod_{j=f}^{|w|-1} b(w_j^{|w|-1})} \end{aligned}$$

where f has the lowest value such that $w_f^{|w|}$ is in the model. Applying the inductive hypothesis to expand $q(w_1^{|w|-1})$ yields

$$\frac{p(w_1^{|w|-1}) \left(\prod_{j=g}^{|w|-1} b(w_j^{|w|-1}) \right) p(w_{|w|} \mid w_f^{|w|-1}) \prod_{j=f}^{|w|} b(w_j^{|w|})}{\prod_{j=f}^{|w|-1} b(w_j^{|w|-1})}$$

where g has the lowest value such that $w_g^{|w|-1}$ is in the model. The backoff terms cancel to yield

$$p(w_1^{|w|-1}) \left(\prod_{j=g}^{f-1} b(w_j^{|w|-1}) \right) p(w_{|w|} \mid w_f^{|w|-1}) \prod_{j=f}^{|w|} b(w_j^{|w|})$$

By construction of g, w_j^{m-1} is not in the model for all $j < g$. Hence, $b(w_j^{m-1}) = 1$ implicitly for all $j < g$. Multiplying by 1,

$$p(w_1^{|w|-1}) \left(\prod_{j=\mathbf{1}}^{f-1} b(w_j^{|w|-1}) \right) p(w_{|w|} \mid w_f^{|w|-1}) \prod_{j=f}^{|w|} b(w_j^{|w|})$$

Recognizing the backoff equation (6.3) to simplify,

$$p(w_1^{|w|-1}) p(w_{|w|} \mid w_{\mathbf{1}}^{|w|-1}) \prod_{j=f}^{|w|} b(w_j^{|w|})$$

Finally, the conditional probability folds as desired

$$q(w_1^{|w|}) = p(w_1^{|w|}) \prod_{j=f}^{|w|} b(w_j^{|w|})$$

□

Entries ending in </s> have backoff one, so it follows from Proposition 4 that sentence-level scores are unchanged.

$$q(\text{<s>}\ w_1^{|w|}\ \text{</s>}) = p(\text{<s>}\ w_1^{|w|}\ \text{</s>})$$

Proposition 4 characterizes q as a pessimistic rest cost for strings that scores sentences in exactly the same way as the baseline using p and b. To save memory, the value q is stored in lieu of p and b. Compared with the baseline, this halves number of values from two to one float per n-gram, except N-grams that already have one value. The impact of this reduction is substantial, as seen in Section 6.4.3. Run-time scoring is also simplified as shown in Algorithm 6 since the language model locates the longest match $w_f^{|w|}$ then returns the value $q(w_{|w|} \mid w_1^{|w|-1}) = q(w_{|w|} \mid w_f^{|w|-1})$ without any calculation or additional lookup. Baseline language models either retrieve backoffs values with additional lookups (Stolcke, 2002; Federico et al., 2008) or carry backoff with state (Section 5.4.3); q effectively moves this step to preprocessing. The disadvantage is that q is not a proper probability and it produces worse rest costs than does the baseline.

In Moses (Koehn et al., 2007) and Joshua (Li et al., 2009), the language model is consulted to score phrases or lexical items in grammar rules as the translation model is loaded. This can be done offline in an embarassingly parallel fashion where memory is not as tight (because the translation model can be streamed). Experiments therefore use an ordinary language model for this initial pass instead of the compressed one. The impact is that translation model pruning (i.e. selecting the top 20 target-side rules by score) and sorting (to prioritize rules for cube pruning) is done with a normal language model. The compressed language model is only used during cube pruning. The cdec (Dyer et al., 2010) decoder does not score rules with the language model as they are loaded.

6.3.3 Combined Scheme

The two language model modifications can be trivially combined by using lower-order probabilities on the left of a string and by charging all backoff penalties on the right of a string. The net result is a language model that uses the same memory as the baseline but has slightly better rest costs.

		Lower			Baseline		
n	Mean	Bias	MSE	Var	Bias	MSE	Var
1	-3.21	.10	.84	.83	$-.12$	.87	.86
2	-2.27	.04	.18	.17	$-.14$	.23	.24
3	-1.80	.02	.07	.07	$-.09$	.10	.09
4	-1.29	.01	.04	.04	$-.10$	.09	.08

Table 6.1: Bias (mean error), mean squared error, and variance (of the error) for the lower-order rest cost and the baseline. Error is the estimated $\log_{10}$ probability minus the final $\log_{10}$ probability. Also shown is the average final $\log_{10}$ probability. Statistics were computed separately for the first word of a fragment ($n = 1$), the second word ($n = 2$), etc. The lower-order estimates are better across the board, reducing error in cube pruning. Statistics were only collected for words in left state.

6.4 Experiments

The test systems are the German-English systems described in Section 5.6.1, one hierarchical and one with target syntax. Language models with orders 1 through 5 were estimated using SRILM[3] (Stolcke, 2002) with interpolated modified Kneser-Ney smoothing (Kneser and Ney, 1995; Chen and Goodman, 1998) and the default pruning settings. In all scenarios, the primary language model has order 5; the other models were used for rest costs. Feature weights were trained with MERT (Och, 2003) on the baseline using a pop limit of 1000 and 100-best output. Since final feature values are unchanged, no retuning was performed. Measurements were collected by running the decoder on the official 3003-sentence test set. These experiments were run on a machine with 64 GB of RAM and 32 cores; this hardware is the same as the 64 GB machine in Chapter 3. Chapters 4 and 5 used different hardware.

6.4.1 Rest Costs as Prediction

Scoring the first few words of a string is a prediction task. The goal is to predict what the probability will be when more context becomes known. Performance on this task was measured by running the decoder on the hierarchical system with a pop limit of 1000. Every time more context became known, the decoder logged[4] the prediction error (estimated log probability minus updated log probability) for both lower-order rest costs and the baseline. Table 6.1 shows the results.

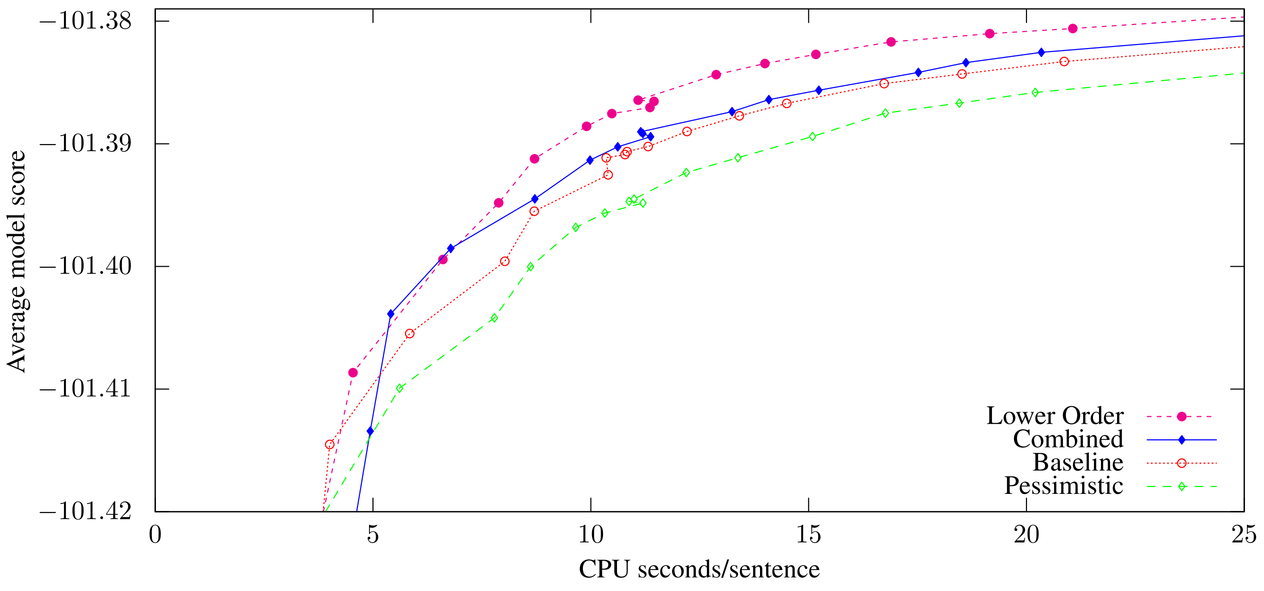
Cube pruning uses relative scores, so bias matters less, though positive bias will favor rules with more arity. Variance matters the most because lower variance means cube pruning's relative rankings are more accurate. Lower-order rest costs are better across the board in terms of absolute bias, mean squared error, and variance. This suggests that lower-order rest costs will make cube pruning more accurate.

6.4.2 Time-Accuracy Trade-Offs

The cube pruning beam size is a trade-off between search accuracy and CPU time. The rest costs improve (or degrade) that trade-off. Search accuracy is measured by the average model score of single-best translations. Model scores are scale-invariant and include a large constant factor; higher is better. Overall performace is also measured with uncased BLEU (Papineni et al., 2002). CPU time is the sum of user and system time used by Moses divided by the number of sentences (3003). Timing includes time to load, though files were forced into the disk cache in advance. Results are shown in Figures 6.2 and 6.1.

[3]This work was performed before Chapter 3.
[4]Logging was only enabled for this experiment.

	Baseline			Lower Order			Pessimistic			Combined		
Pop	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
2	2.96	−101.85	21.19	2.44	−101.80	21.63	2.71	−101.90	20.85	3.05	−101.84	21.37
10	2.80	−101.60	21.90	2.42	−101.58	22.20	2.95	−101.63	21.74	2.69	−101.60	21.98
50	3.02	−101.47	22.18	3.11	−101.46	22.34	3.46	−101.48	22.08	2.67	−101.47	22.14
690	10.83	−101.39	22.28	11.45	−101.39	22.25	10.88	−101.40	22.25	11.19	−101.39	22.23
900	13.41	−101.39	22.27	14.00	−101.38	22.24	13.38	−101.39	22.25	14.09	−101.39	22.22
1000	14.50	−101.39	22.27	15.17	−101.38	22.25	15.09	−101.39	22.26	15.23	−101.39	22.23
1350	18.52	−101.38	22.27	19.16	−101.38	22.23	18.46	−101.39	22.25	18.61	−101.38	22.23
5000	59.67	−101.38	22.24	61.41	−101.38	22.22	59.76	−101.38	22.27	61.38	−101.38	22.22



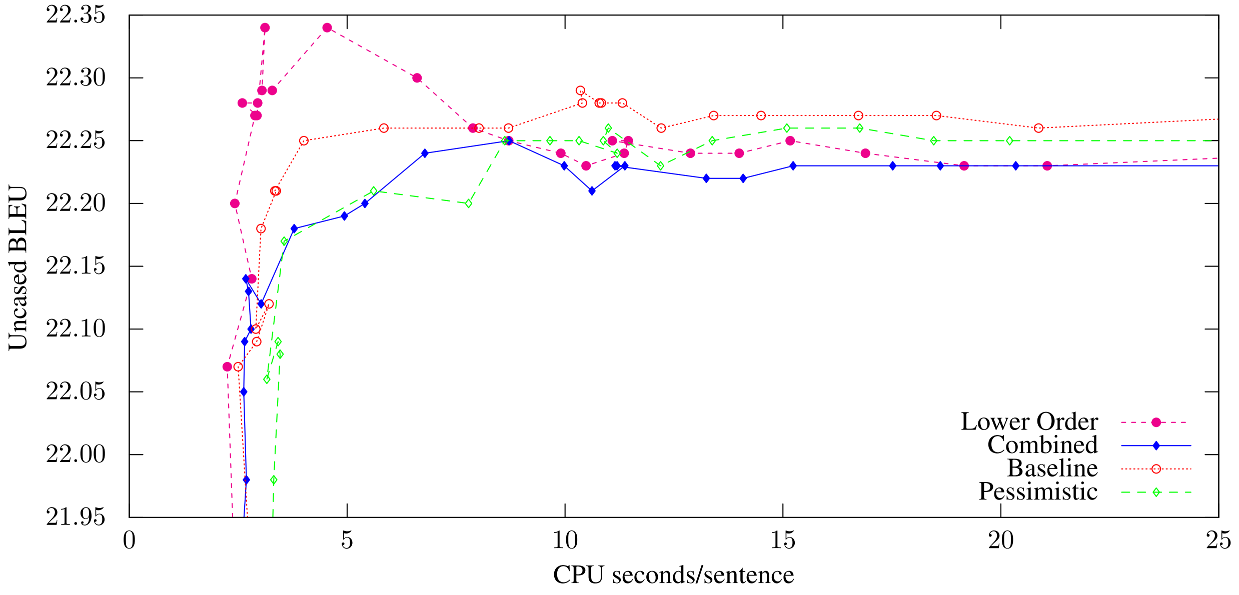
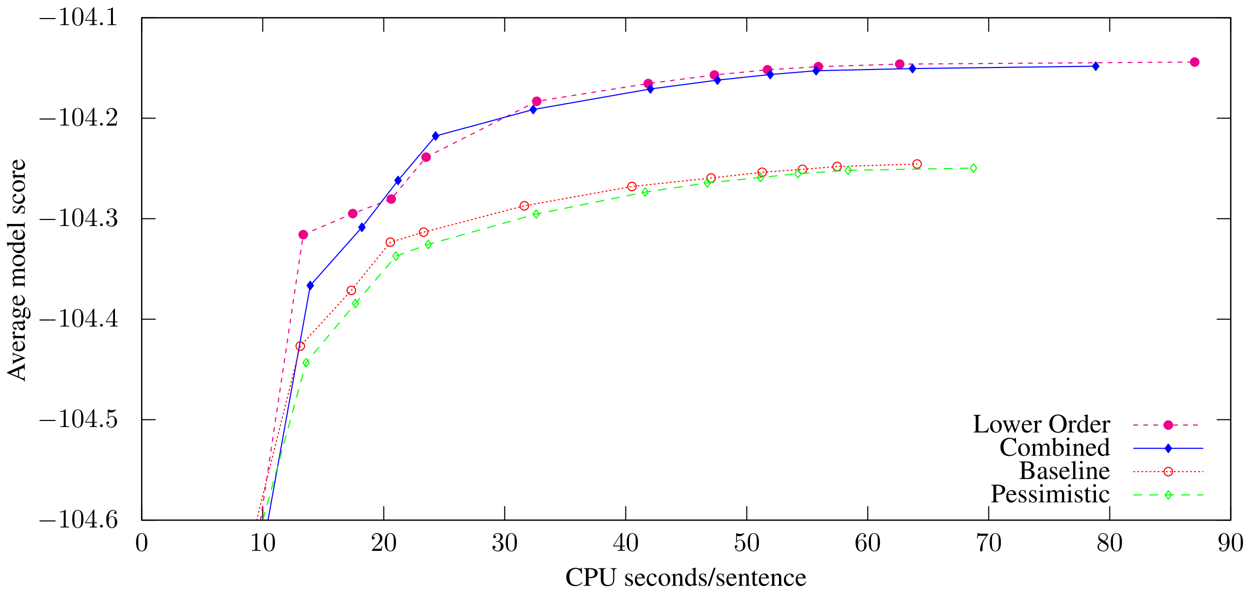


Figure 6.1: Hierarchical system performance. Select beam sizes are shown in the table.

	Baseline			Lower Order			Pessimistic			Combined		
Pop	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
2	3.29	−105.56	20.45	3.68	−105.44	20.79	3.74	−105.62	20.01	3.18	−105.49	20.43
10	5.21	−104.74	21.13	5.50	−104.72	21.26	5.43	−104.77	20.85	5.67	−104.75	21.10
50	23.30	−104.31	21.36	23.51	−104.24	21.38	23.68	−104.33	21.25	24.29	−104.22	21.34
500	54.61	−104.25	21.33	55.92	−104.15	21.38	54.23	−104.26	21.31	55.74	−104.15	21.40
700	64.08	−104.25	21.34	87.02	−104.14	21.42	68.74	−104.25	21.29	78.84	−104.15	21.41



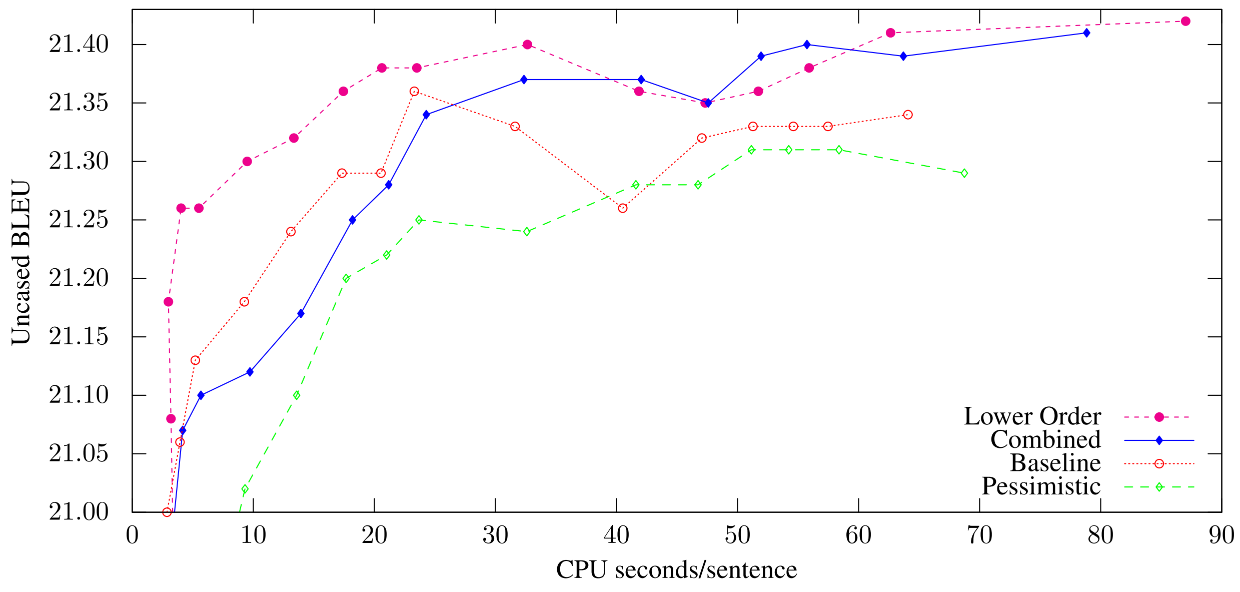


Figure 6.2: Target-syntax performance. Select beam sizes are shown in the table.

Lower-order rest costs perform better in both systems, reaching plateau model scores and BLEU with less CPU time. In the hierarchical system, peak BLEU 22.34 is achieved under the lower-order condition with pop limits 50 and 200, while other scenarios are still climbing to the plateau. With a pop limit of 1000, the baseline's average model score is −101.3867. Better average models scores are obtained from the lower-order model with pop limit 690 using 79% of baseline CPU, the combined model with pop limit 900 using 97% CPU, and the pessimistic model with pop limit 1350 using 127% CPU.

The gain is much larger for target syntax, where a pop limit of 50 outperforms the baseline with pop limit 700. CPU time per sentence is reduced to 23.5 seconds from 64.0 seconds, a 63.3% reduction. The combined setting, using the same memory as the baseline, shows a similar 62.1% reduction in CPU time. This difference appears to be due to improved grammar rule scoring that impacts pruning and sorting. In the target syntax model, the grammar is not saturated (i.e. less pruning will still improve scores) but pruning is nonetheless used for tractability reasons. The lower-order rest costs are particularly useful for grammar pruning because lexical items are typically less than five words long (and frequently only word).

Pessimistic compression does worsen search, requiring 27% more CPU in the hierarchical system to achieve the same quality. This is worthwhile to fit large-scale language models in memory, especially if the alternative is a remote language model.

Further experiments with improved rest costs appear in Chapter 7.

6.4.3 Memory Usage

Rest costs add a value (for lower-order probabilities) or remove a value (pessimistic compression) for each n-gram except those of highest order ($n = N$). The combined condition adds one value and removes another, so it uses the same memory as the baseline. The memory footprint of adding or removing a value depends on the number of such n-grams, the underlying data structure, and the extent of quantization.

Values are reported for the test language model which has 135 million n-grams for $n < 5$ and 56 million 5-grams. Actual memory usage is reported for the data structures in Chapter 4. Theoretical values are also reported for minimal perfect hashing from ShefLM (Guthrie and Hepple, 2010). Minimal perfect hashing assumes the Compress, Hash and Displace algorithm (Belazzougui et al., 2008) with 8-bit signatures and 8-bit quantization. Table 6.2 shows the results. Storage size of the smallest model is reduced by 26%, bringing higher-quality smoothed models in line with stupid backoff models that also store one value per n-gram.

Structure	Baseline	Change	%
Probing	4,072	517	13%
Trie	2,647	506	19%
8-bit quantized trie	1,236	140	11%
8-bit minimal perfect hash	540	140	26%

Table 6.2: Size in megabytes of the test language model, excluding operating system overhead. Change is the cost of adding an additional value to store lower-order probabilities. Equivalently, it is the savings from pessimistic compression.

6.5 Summary

Two techniques have been presented that trade between accuracy and memory usage. Efficiently storing lower-order probabilities and using them as rest costs improves both cube pruning (21% CPU reduction in a hierarchical system) and better grammar pruning (net 63% CPU time reduction with target syntax) at the expense of 13–26% more RAM for the language model. This grammar pruning improvement is surprising both in the impact relative to changing the pop limit and simplicity of implementation, since it can be done offline. Compressing the language model to halve the number of values per n-gram (except N-grams) results in a 13–26% reduction in RAM with 26% over the smallest model, costing 27% more

CPU and leaving overall sentence scores unchanged. This compression technique is likely to have more general application outside of machine translation, especially where only sentence-level scores are required.

The next chapter presents a new search algorithm. Just as with cube pruning, improved rest costs are optional. However, the gains from improved rest costs are even larger, as seen in the next chapter.

Chapter 7

Searching Hypergraphs[1]

It is equally bad when one speeds on the guest unwilling to go, and when he holds back one who is hastening. Rather one should befriend the guest who is there, but speed him when he wishes. –Homer, *The Odyssey*

Searching a hypergraph for high-scoring hypotheses is a common subproblem in many statistical natural language processing systems (Section 2.4.2). When the score includes an N-gram language model, search is computationally difficult because the language model examines surface n-grams without regard to the hypergraph's structure. Cube pruning (Chiang, 2007) is the most popular approach to this problem and is used in previous chapters. This chapter introduces a new search algorithm designed to improve upon cube pruning. Experiments show that the new algorithm makes machine translation decoding 1.15 to 6.60 times as fast as with cube pruning for beam sizes 20 and above.

7.1 Introduction

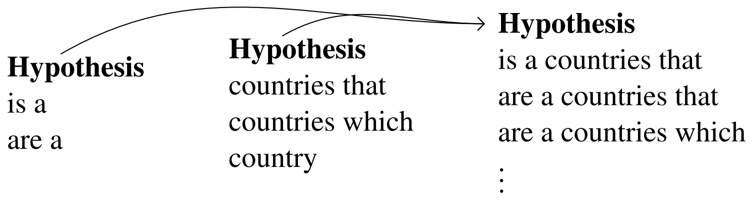


Figure 7.1: Cube pruning (Chiang, 2007) has no notion that hypotheses end with "a" or begin with "countries", so it generates many disfluent hypotheses containing "a countries".

Competing hypotheses often have words in common because they translate the same source span, transcribe the same speech, or recognize the same characters. Hypothesis boundary words matter to the language model because these can participate in N-grams when the hypothesis grows by concatenation, a notion formalized in Chapter 5 as state. When hypotheses have exactly the same state, they can be recombined and efficiently handled by algorithms such as cube pruning (Chiang, 2007). However, many hypothesis states share some, but not all, words. Cube pruning makes no effort to exploit these common words. An example of the problem is shown in Figure 7.1. The language model is likely to score hypotheses similarly if their

[1]A prior version of this work was published as Heafield et al. (2013a).

outermost words are the same. This chapter contributes a new search algorithm that exploits these observations by recursively grouping hypotheses according to common prefixes and suffixes. An example is shown in Figure 7.2.

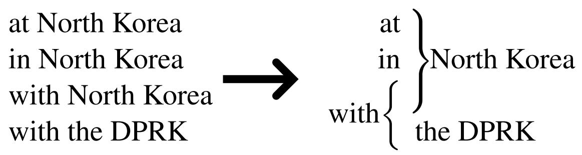


Figure 7.2: Hypotheses are grouped by common prefixes and suffixes.

The search algorithm is based on beam search (Lowerre, 1976; Chiang, 2007). Introduced in Section 2.4.2, beam search visits hypergraph vertices in bottom-up (topological) order. Within a vertex, it selects up to k competing hypotheses (i.e. translations of the same source words with the same grammar label). These can later be used by other vertices to build larger hypotheses. The parameter k is a time-accuracy trade-off: larger k increases both CPU time (because more options are considered when the vertex is used as an antecedent) and accuracy (because more hypotheses are considered).

Ordinary beam search (Section 2.4.2) generates all possible hypotheses for a vertex then keeps only the top k by score. Cube pruning is based on beam search but uses a priority queue to select k hypotheses to generate instead of generating all of them. This work contributes another method, also based on a priority queue, that generates k hypotheses. The difference is that the priority queue contains groups of hypotheses and these groups are refined in a best-first fashion. Eventually, some groups are so refined that they contain a single hypothesis. The algorithm stops when it has extracted k groups that each contain a single hypothesis.

7.2 Related Work

7.2.1 Alternatives to Bottom-Up Search

Left-to-right decoding (Watanabe et al., 2006; Huang and Mi, 2010) builds output strings from left to right. Therefore, every hypothesis is bound to the beginning of sentence and has empty left language model state (Chapter 5). Recombination happens more often because only right language model state matters. The disadvantage is that hypergraph vertices are visited multiple times in different language model contexts, whereas this work visits each hypergraph vertex exactly once. In principle, this work is compatible with left-to-right decoding; because left state is empty, hypotheses will naturally only be grouped by common suffixes in their right state.

Cube growing (Huang and Chiang, 2007) generates hypotheses on top-down demand (Huang and Chiang, 2007). However, cube growing showed little improvement in Moses (Xu and Koehn, 2012) and is not supported by the current version. Jane (Vilar et al., 2010) does implement cube growing, but is slower than Moses (Section 2.4.3) and their cube growing implementation is missing optimizations, including Chapter 5, so the comparison would be unfair. Cube pruning keeps buffers of hypotheses in each vertex and these buffers could be grouped by applying this work.

7.2.2 Baseline: Cube Pruning

Cube pruning (Chiang, 2007) is a fast approximate search algorithm and the primary baseline. It chooses k hypotheses by popping them off the top of a priority queue. Initially, the queue is populated with hypotheses made from the best (highest-scoring) parts. These parts are an edge and a hypothesis from each vertex referenced by the edge (also known as tails). When a hypothesis is popped, several next-best alternatives are pushed. These alternatives substitute the next-best edge or a next-best hypothesis from one of the vertices.

This work follows a similar pattern of popping one queue entry then pushing multiple entries. However, queue entries in this work are a group of hypotheses while cube pruning's entries are a single hypothesis.

Hypotheses are usually fully scored before being placed in the priority queue. As a consequence, cube pruning calls the language model more than k times, because some pushed hypotheses are never popped. An alternative is to score hypotheses after they are popped from the queue, so that the language model is called k times. In this alternative, hypotheses in the queue are prioritized by their *additive score*. The additive score is the edge's score plus the score of each component hypothesis (non-terminal), ignoring the non-additive aspect of the language model. Additive cube pruning is implemented as an option in Moses (Hoang et al., 2009) and appears as a baseline whenever Moses is used. At a high level, this work is a compromise between both options: hypothesis groups start with additive scores and incrementally transition to full scores as words are revealed.

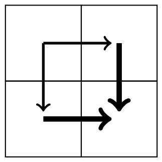


Figure 7.3: In normal cube pruning, multiple neighbors propose the same alternative hypothesis. This example shows a rule with two non-terminals, each of which has two possible values. The top-left option combines the best non-terminals. The bottom-left and top-right options substitute a single-best alternative. The bottom-right option is a next-best alternative to both the top-right and bottom-left hypotheses.

Gesmundo and Henderson (2010) noted that cube pruning produces duplicate queue entries. An example is shown in Figure 7.3: there are multiple ways to reach the same hypothesis. Normal cube pruning addresses this issue with a hash table that drops duplicate hypotheses before they are pushed onto the queue. Gesmundo and Henderson (2010) presented two ways to modify the algorithm to prevent duplicate queue entries, eliminating the need for a hash table. They implemented both versions in cdec (Dyer et al., 2010) and their work appears as baselines when cdec is used.

Hopkins and Langmead (2009) characterized cube pruning as A* search (Hart et al., 1968) with an inadmissible heuristic. Their analysis showed deep and unbalanced search trees. This work can be interpreted as a partial rebalancing of the search trees.

7.2.3 Exact Algorithms

A number of exact search algorithms have been developed (Rush and Collins, 2011; Iglesias et al., 2011; Aziz et al., 2013). None of these tractably scales to the size of hypergraphs and language models used in many modern machine translation systems (Bojar et al., 2013b).

Rush and Collins (2011) compile the hypergraph and language model into an integer linear program. They then optimize by taking the dual and solving by Lagrangian relaxation. If the dual has an optimal integer solution, as it does in most cases, then a single-best translation is recovered. However, the work only dealt with language models up to order three. Almost all systems in the 2013 Workshop on Machine Translation (Bojar et al., 2013b) used models of order four or five, where the search problem is exponentially harder. One could exactly decode with a language model of order three, but doing so simply moves approximation from search to the objective function.

Iglesias et al. (2011) represent the search space as a recursive transition network and the language model as a weighted finite state transducer. Using standard finite state algorithms, they intersect the two automatons then exactly search for the highest-scoring paths. However, the intersected automaton is too large. The authors suggested removing low probability entries from the language model, but this form of pruning negatively impacts translation quality (Moore and Quirk, 2009; Chelba et al., 2010). Their work bears some similarity to the present work in that partially overlapping state will be collapsed and efficiently handled together. The key difference is that their work performs full intersection before grouping hypotheses together with scores. The key advantage of this work is that groups have a score which can be used for pruning *before* the group is expanded, enabling pruning without first constructing the intersected automaton.

Published after this work, Aziz et al. (2013) improve upon Iglesias et al. (2011) by iteratively refining the language model. Initially, the language model contains only unigrams. These unigrams are weighted by upper bounds from a full-order language model (Carter et al., 2012), effectively relaxing the model. They intersect the relaxed model with the search space, and extract a single-best hypothesis similar to Iglesias et al. (2011). In this hypothesis, they identify the most violated constraint, namely the word whose probability bound is furthest from its true conditional probability in the sentence. The language model is then refined by replacing that word's unigram probability with bigram probabilities for every possible context. The process repeats until a sentence is extracted whose probability matches the upper bounds. Subsequent iterations can refine a bigram to trigrams, etc. The problem is that their method is very expensive, in one case taking seven hours to translate a seven-word sentence.

In contrast to Aziz et al. (2013) and their prior work Carter et al. (2012), the present work is approximate and uses average-case probabilities. If upper bounds are used with this work, then the algorithm is still approximate due to beam search and, empirically, the time-accuracy trade-off worsens. Another difference is that the present work locally refines probabilities instead of updating them in the entire hypergraph. This work is also primarily concerned with one vertex at a time while their approach operates globally. The open question is whether global reasoning can be done cheaply.

7.2.4 Coarse-to-Fine

Coarse-to-fine (Zhang and Gildea, 2008; Petrov et al., 2008) performs multiple pruning passes, each time with more detail. Search is a subroutine of coarse-to-fine and this work is inside search, so the two are compatible. There are several forms of coarse-to-fine search; the closest to this work increases the language model order each iteration. The key difference is that the present work operates inside search, so it can handle hypotheses at different levels of refinement and use scores to choose where to further refine hypotheses. Coarse-to-fine decoding cannot do this because models are refined in lock step. As Petrov et al. (2008) point out, coarse-to-fine decoding works best with word classes, suggesting that this work might also be improved by using word classes as intermediate steps in refinement.

7.3 The New Algorithm

The primary idea is to group hypotheses with similar language model state. These groups are entries in a priority queue. When a group reaches the top of the queue, the algorithm checks whether the group contains a single hypothesis. If so, the group is removed from the queue and the hypothesis is output. Otherwise, the group is removed from the queue and split into two smaller groups, both of which are pushed onto the queue. The process repeats until k hypotheses have been extracted or the priority queue is empty.

The following sections formalize what these groups are (partial state), that the groups have a recursive structure (state tree), how groups are split (bread crumbs), using groups with hypergraph edges (partial edge), prioritizing search (scoring), and best-first search (priority queue).

All of these are components of an algorithm that generates k hypotheses within a hypergraph vertex. Beam search calls this algorithm for each hypergraph vertex in bottom-up order. The algorithm is inductive:

each vertex assumes that vertices below have already run the algorithm and extracted k hypotheses.

7.3.1 Partial State

Recalling Chapter 5 and Table 5.1 in particular, the state of string "the few nations that have diplomatic relations with North Korea" might be denoted (the few $\dashv \diamond \vdash$ Korea) where "the few" is left state and "Korea" is right state.

The algorithm is based on *partial state*. Partial state is simply state with more inner words elided. For example, (the $\diamond$ Korea) is a partial state for (the few $\dashv \diamond \vdash$ Korea). Empty state is denoted using the customary symbol for the empty string, ϵ. For example, ($\epsilon \diamond \epsilon$) is the empty partial state. Terminators $\dashv$, $\vdash$, $\lhd$, and $\rhd$ can be elided just like words.

The terminators serve two purposes. First, they distinguish whether state is short because state minimization applied, denoted $\dashv$ and $\vdash$, or because the string has too few words, denoted $\lhd$ and $\rhd$. These distinctions impact scoring as explained in Chapter 5. Second, states can have different lengths and the terminators act like alternative words. For example, the hypothetical string "the few countries that have relations with periwinkle Korea" might have state (the few $\dashv \diamond \vdash$ Korea) because the language model knows that "periwinkle Korea" will never extend to the right. With the same language model, the string "the few countries that have relations with North Korea" might have state (the few $\dashv \diamond \vdash$ North Korea). The partial state (the few $\diamond$ Korea) reasons over both strings, which it can do because it lacks a terminator. When this partial state is refined, "$\vdash$" acts as an alternative to "North".

7.3.2 State Tree

States (the few $\dashv \diamond \vdash$ Korea) and (the $\dashv \diamond \vdash$ Korea) have words in common, so the partial state (the $\diamond$ Korea) can be used to reason over both of them. Transitively, ($\epsilon \diamond \epsilon$) reasons over all partial states. Formally, partial states can be placed into a hierarchy of equivalence classes.

The *state tree* is a hierarchy of partial states. The root of the tree is the empty partial state ($\epsilon \diamond \epsilon$) that reasons over all hypotheses. From the root, the tree branches by the first word of state, the last word, the second word, the penultimate word, and so on. If left or right state is exhausted, then branching continues using the remaining state. The branching order prioritizes the outermost words because these can be used to update the language model probability. The decision to start with left state is arbitrary; this is done to create a single branching order. An example tree is shown in Figure 7.4.

When beam search is complete, every vertex of the hypergraph has a state tree containing up to k hypotheses. This leads to some unfortunate overlap of terminology; to disambiguate, the terms *vertex* and *edge* refer to the hypergraph while the term *node* refers to the state tree.

As an optimization, nodes of the state tree reveal prefixes and suffixes that all of its descendants agree upon. For example, in Figure 7.4, all hypotheses that begin with "the" also end with "Korea". The node (the $\diamond$ ϵ) recognizes that this is the case and updates itself to become (the $\diamond$ Korea). As a result, no node has a single child and branching happens only when there is a decision to be made. The original tree is shown in Figure 7.4 and the optimized version is shown in Figure 7.5. The optimization applies only if all descendants agree; because one hypothesis ends with "DPRK", the root node could not reveal that (the $\diamond$ Korea) and (a $\dashv \diamond$ Korea) both end in "Korea".

The state tree is built lazily as shown in Figure 7.6. Initially, the root node simply has an unstructured array of hypotheses. When its children are first needed, the root node groups the hypotheses, constructs a child node for each group, and hands each hypothesis over to the appropriate child. In turn, each of these children initially holds an array of hypotheses and will only construct its children when necessary. This saves CPU time because low-scoring nodes may never construct their children.

$$\begin{tikzcd}
 & & & (\text{the few} \diamond \text{Korea}) \arrow[r] & (\text{the few} \diamond \vdash \text{Korea}) \arrow[r] & (\text{the few} \dashv \diamond \vdash \text{Korea}) \\
 & (\text{the} \diamond \epsilon) \arrow[r] & (\text{the} \diamond \text{Korea}) \arrow[ur] \arrow[dr] & & & \\
 & & & (\text{the} \dashv \diamond \text{Korea}) \arrow[r] & (\text{the} \dashv \diamond \vdash \text{Korea}) & \\
(\epsilon \diamond \epsilon) \arrow[uur] \arrow[r] \arrow[ddr] & (\text{some} \diamond \epsilon) \arrow[r] & (\text{some} \diamond \text{DPRK}) \arrow[r] & (\text{some} \dashv \diamond \text{DPRK}) \arrow[r] & (\text{some} \dashv \diamond \vdash \text{DPRK}) & \\
 & & & & (\text{a} \dashv \diamond \text{in Korea}) \arrow[r] & (\text{a} \dashv \diamond \vdash \text{in Korea}) \\
 & (\text{a} \diamond \epsilon) \arrow[r] & (\text{a} \diamond \text{Korea}) \arrow[r] & (\text{a} \dashv \diamond \text{Korea}) \arrow[ur] \arrow[dr] & & \\
 & & & & (\text{a} \dashv \diamond \vdash \text{Korea}) &
\end{tikzcd}$$

Figure 7.4: A state tree containing five states: (the few ⊣ ◇ ⊢ Korea), (the ⊣ ◇ ⊢ Korea), (some ⊣ ◇ ⊢ DPRK), (a ⊣ ◇ ⊢ in Korea), and (a ⊣ ◇ ⊢ Korea). Nodes of the tree are partial states. The branching order is the first word, the last word, the second word, and so on. If the left or right state is exhausted, then branching continues with the remaining state. For purposes of branching, termination symbols ⊣ and ⊢ act like normal words. This ensures that short states (◁ and ▷) follow a different branch after state has been revealed.

$$\begin{tikzcd}
 & & (\text{the few} \dashv \diamond \vdash \text{Korea}) \\
 & (\text{the} \diamond \text{Korea}) \arrow[ur] \arrow[dr] & \\
 & & (\text{the} \dashv \diamond \vdash \text{Korea}) \\
(\epsilon \diamond \epsilon) \arrow[uur] \arrow[rr] \arrow[ddr] & & (\text{some} \dashv \diamond \vdash \text{DPRK}) \\
 & & (\text{a} \dashv \diamond \vdash \text{in Korea}) \\
 & (\text{a} \dashv \diamond \text{Korea}) \arrow[ur] \arrow[dr] & \\
 & & (\text{a} \dashv \diamond \vdash \text{Korea})
\end{tikzcd}$$

Figure 7.5: The optimized version of Figure 7.4. Nodes immediately reveal the longest shared prefix and suffix among hypotheses below them.

1

$$\begin{tikzcd}
(\epsilon \diamond \epsilon)
\end{tikzcd}$$

2

$$\begin{tikzcd}
 & (\text{the} \diamond \text{Korea}) \\
(\epsilon \diamond \epsilon) \arrow[ur] \arrow[r] \arrow[dr] & (\text{some} \dashv \diamond \vdash \text{DPRK}) \\
 & (\text{a} \dashv \diamond \text{Korea})
\end{tikzcd}$$

3

$$\begin{tikzcd}
 & & (\text{the few} \dashv \diamond \vdash \text{Korea}) \\
 & (\text{the} \diamond \text{Korea}) \arrow[ur] \arrow[dr] & \\
 & & (\text{the} \dashv \diamond \vdash \text{Korea}) \\
(\epsilon \diamond \epsilon) \arrow[uur] \arrow[rr] \arrow[ddr] & & (\text{some} \dashv \diamond \vdash \text{DPRK}) \\
 & & \\
 & (\text{a} \dashv \diamond \text{Korea}) &
\end{tikzcd}$$

Figure 7.6: The state tree in Figure 7.5 is built lazily as children are needed. This is more efficient because some children may never be built.

Each node has a score. For leaves, this score is copied from the underlying hypothesis (or best hypothesis if some other feature prevented recombination). The score of an internal node is the maximum score of its children. As an example, the root node's score is the same as the highest-scoring hypothesis in the tree. Children are sorted by score.

7.3.3 Bread Crumbs

The state tree is explored in a best-first manner. Specifically, when the algorithm visits a node, it considers that node's best child. The best child reveals more words, so the score may go up or down when the language model is consulted. Therefore, simply following best children may lead to a poor hypothesis. Some backtracking mechanism is required; this functionality is implemented with bread crumbs. A bread crumb is a note to return to a node and consider the other children. Thus, when a node is visited, it is split into to parts: the best remaining child and a bread crumb. The bread crumb encodes the node that was visited and how many children have already been considered. Figure 7.7 shows an example.

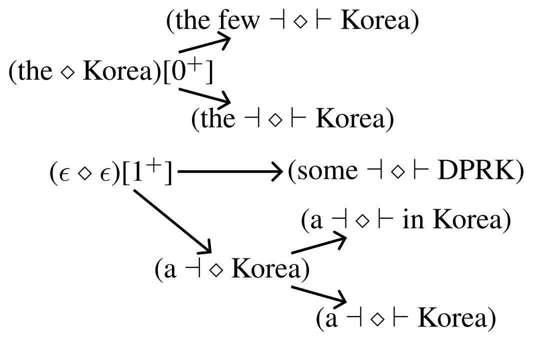


Figure 7.7: Visiting the root node partitions the tree into best child (the $\diamond$ Korea)$[0^+]$ and bread crumb $(\epsilon \diamond \epsilon)[1^+]$. The original data structure (Figure 7.5) remains intact for use elsewhere.

More formally, each node has an array of children sorted by score, so it suffices for the bread crumb to keep an index in this array. An index of zero denotes that no child has been visited. Continuing the example from Figure 7.5, $(\epsilon \diamond \epsilon)[0^+]$ denotes the root partial state with children starting at index 0 (i.e. all of them). Visiting $(\epsilon \diamond \epsilon)[0^+]$ yields best child (the $\diamond$ Korea)$[0^+]$ and bread crumb $(\epsilon \diamond \epsilon)[1^+]$ as shown in Figure 7.7. Later, the search algorithm may return to $(\epsilon \diamond \epsilon)[1^+]$, yielding best child (some $\dashv \diamond \vdash$ DPRK)$[0^+]$ and bread crumb $(\epsilon \diamond \epsilon)[2^+]$. If there is no remaining sibling, there is no bread crumb and only the reamining child is produced.

The index restricts the children to those with that index or above. More formally, the hypotheses below a node are those of its children

$$\text{hypotheses}(\text{node}) = \bigsqcup_{i=0}^{|\text{node}|-1} \text{hypotheses}(\text{node}[i])$$

where $\sqcup$ takes the union of disjoint sets and node$[i]$ is the ith child. In a bread crumb annotated with index c, only descendents by the remaining children are considered

$$\text{hypotheses}(\text{node}[c^+]) = \bigsqcup_{i=c}^{|\text{node}|-1} \text{hypotheses}(\text{node}[i])$$

It follows that the set of descendant hypotheses is *partitioned* into two disjoint sets

$$\text{hypotheses}(\text{node}[c^+]) = \text{hypotheses}(\text{node}[c]) \bigsqcup \text{hypotheses}(\text{node}[c+1^+])$$

Therefore, exactly the same hypotheses are considered when a node is split; they have simply been partitioned into two sets. Continuing the example of Figure 7.5, the root node $(\epsilon \diamond \epsilon)$ has hypotheses

$$\text{hypotheses}((\epsilon \diamond \epsilon)) = \{(\text{the few} \dashv \diamond \vdash \text{Korea}), (\text{the} \dashv \diamond \vdash \text{Korea}), \\ (\text{some} \dashv \diamond \vdash \text{DPRK}), (\text{a} \dashv \diamond \vdash \text{in Korea}), (\text{a} \dashv \diamond \vdash \text{Korea})\}$$

These are partitioned into

$$\text{hypotheses}((\text{the} \diamond \text{Korea})) = \{(\text{the few} \dashv \diamond \vdash \text{Korea}), (\text{the} \dashv \diamond \vdash \text{Korea})\}$$

and

$$\text{hypotheses}((\epsilon \diamond \epsilon)[1^+]) = \{(\text{some} \dashv \diamond \vdash \text{DPRK}), (\text{a} \dashv \diamond \vdash \text{in Korea}), (\text{a} \dashv \diamond \vdash \text{Korea})\}$$

as illustrated in Figure 7.7.

7.3.4 Partial Edge

As mentioned in the introduction, the short-term goal of the algorithm is to select high-scoring hypotheses for a given hypergraph vertex. The hypergraph vertex has a number of outgoing (downward) hyperedges, all of which can be applied to form hypotheses. Each hypergraph edge is a string comprised of words and references to vertices (in parsing, terminals and non-terminals). A hypergraph edge is converted to a *partial edge* by replacing each vertex reference with the root node from that vertex. For example, the hypergraph edge "is v ." referencing vertex v becomes partial edge "is $(\epsilon \diamond \epsilon)[0^+]$.". In general, there can be references to multiple vertices, such as "u one of v" referencing vertices u and v. Each reference is replaced by the root node from the respective vertex. For machine translation, it is worth noting that only the target-side rule in target language order matters; all necessary source information was encoded by the parser when it built the hypergraph.

Partial edges contain bread crumbs, each of which represents a set of hypotheses. A partial edge efficiently reasons over all possible combinations of these hypotheses. *Partitioning* a partial edge divides that set into two. Specifically, a heuristic picks one of the bread crumbs, splits it in two, and substitutes these values into the partial edge. Continuing the example from Figure 7.5, "is $(\epsilon \diamond \epsilon)[0^+]$." partitions into "is (the $\diamond$ Korea)$[0^+]$." and "is $(\epsilon \diamond \epsilon)[1^+]$."

Formally, partitioning proceeds as follows. A heuristic chooses one of the bread crumbs to partition, skipping any that contain a single hypothesis. If all of the bread crumbs contain a single hypothesis, then the partial edge is a complete hypothesis and it is placed into the beam. Otherwise, the heuristic picks the bread crumb with the fewest words revealed. As a tie breaker, it chooses the leftmost bread crumb. The chosen bread crumb is split, yielding the best child and another bread crumb that encodes the alternatives as described in the previous section. Two copies of the partial edge are made, one with the best child and one with the alternative.

7.3.5 Scoring

Every partial edge has a score that determines its search priority. This section explains where these scores come from. The central idea is that language model scores are estimated whenever context is indeterminate. As words are revealed, these estimated scores are updated to account for known language model context.

Hypergraph edges have a score that includes additive features, such as length and log probabilities from the translation model, and an estimated log language model probability for any words that appear in the rule. Words in an edge usually have insufficient context to be completely scored by the language model, because they are at the start of the edge or are preceded by a non-terminal. Chapter 6 described several ways in which these words can be scored. This work supports all of the methods described in Chapter 6, including the baseline. For notational convenience, the function r abstractly refers to some method of estimating language model scores. This function can be the lower-order rest costs of Chapter 6, the baseline ($r = p$), the pessimistic value q from Section 6.1.2, upper bounds (Huang et al., 2012; Carter et al., 2012), or any other estimate. At a higher level, the function is used to score edges; for example, edge "is v ." incorporates estimate

$$\log r(\text{is})r(.)$$

into its score. The same applies to hypotheses: (the few $\dashv \diamond \vdash$ Korea) includes estimate

$$\log r(\text{the})r(\text{few} \mid \text{the})$$

because the words in left state are those with insufficient context.

The score of a bread crumb is the maximum score of its descendants as defined in Section 7.3.3. For example, the bread crumb $(\epsilon \diamond \epsilon)[1^+]$ has a lower score than $(\epsilon \diamond \epsilon)[0^+]$ because the best child (the $\diamond$ Korea)$[0^+]$ and its descendants no longer contribute to the maximum.

The score of partial edge "is $(\epsilon \diamond \epsilon)[0^+]$." is the sum of scores from its parts: edge "is v ." and bread crumb $(\epsilon \diamond \epsilon)[0^+]$. In general, there may be zero or multiple bread crumbs in a partial edge; their scores are all summed. The edge's score includes estimated log probability $\log r(\text{is})r(.)$ as explained earlier. The bread crumb's score comes from its highest-scoring descendent (the few $\dashv \diamond \vdash$ Korea) and therefore includes estimate $\log r(\text{the})r(\text{few} \mid \text{the})$.

Estimates are updated as words are revealed. Continuing the example, "is $(\epsilon \diamond \epsilon)[0^+]$." has best child "is (the $\diamond$ Korea)$[0^+]$." In this best child, the estimate $r(.)$ is updated to $r(. \mid \text{Korea})$. Similarly, $r(\text{the})$ is replaced with $r(\text{the} \mid \text{is})$. Updates examine only words that have been revealed: $r(\text{few} \mid \text{the})$ remains unrevised.

Updates are computed efficiently by using pointers as described in Chapter 5. To summarize, the language model computes

$$\frac{r(\text{the} \mid \text{is})}{r(\text{the})}$$

in a single call. The term is efficient to retrieve because the search algorithm retained a pointer to the entry for "the". Multiplying by the term (adding in log space) effectively replaces $r(\text{the})$ with $r(\text{the} \mid \text{is})$ as desired in the previous paragraph.

7.3.6 Priority Queue

The algorithm searches in best-first order. This is controlled by a priority queue containing partial edges. An example is shown in Table 7.1.

Partial Edge	**Score**
is a (countries that $\diamond$ Korea .)$[0^+]$ </s>	−3.2
$(\epsilon \diamond \epsilon)[1^+]$ the $(\epsilon \diamond \epsilon)[0^+]$	−4.5
are a country	−4.7
is a $(\epsilon \diamond \epsilon)[1^+]$ </s>	−6.2

Table 7.1: Example partial edges in a priority queue. There can be multiple ways to derive the same vertex, including partial edges with different numbers of bread crumbs.

The queue is populated by converting all outgoing hypergraph edges into partial edges and pushing them onto the queue. After this initialization, the algorithm loops. Each iteration begins by popping the top-scoring partial edge off the queue. If all of the bread crumbs contain a single hypothesis, then the partial edge is converted to a hypothesis and placed in the beam. Otherwise, the partial edge is partitioned as described in Section 7.3.3. The two resulting partial edges are pushed onto the queue. Looping continues with the next iteration until the queue is empty or the beam is full. After the loop terminates, the beam is given to the root node of the state tree; other nodes will be built lazily as described in Section 7.3.2.

7.3.7 Overall Algorithm

The preceding sections described how k hypotheses are selected within a hypergraph vertex. Overall, the algorithm visits hypergraph vertices in bottom-up order, just like beam search and cube pruning (Chiang, 2007). The algorithm runs in each vertex, making use of state trees in vertices below. The top of the tree contains full hypotheses. If a K-best list is desired, then recombination (when language model states are completely equal) and extraction work the same way as with cube pruning (Chiang, 2007). Specifically, hypotheses remember up to K lower-scoring paths that recombined. At the end, these hypotheses can be unpacked exactly in a greedy manner because recombination ensures that all scores are additive.

7.4 Experiments

7.4.1 Scenarios

Experiments cover four language pairs, three grammar formalisms, two decoder implementations, and two ways to estimate probability given incomplete context. Due to the size of the results, details appear in Appendix A; this section presents high-level summaries.

Moses experiments use the five machine translation systems described in Section 5.6.1. This chapter also experiments with cdec (Dyer et al., 2010) using two systems: a port of the German–English hierarchical system from Moses (with comparable model scores) and a French–English translation system.

The French–English system, built and described by Ammar et al. (2013), uses syntactic labels on both source and target sides of the synchronous context free grammar. This complements the existing set of systems that have very simple labels (X and S) on both sides (hierarchical) or on the source side (target syntax). The French–English system was submitted by Ammar et al. (2013) to the 2013 Workshop on Machine Translation (Bojar et al., 2013a) as CMU-TREE-TO-TREE and uses constrained data provided by the workshop. The language model has order N=4. Experiments are run on the official 3000-sentence test set.

Experiments, including baselines, use the fast linear probing data structure from Chapter 4 and all state optimizations from Chapter 5. Chapter 6 contributed different ways to score words that have incomplete context. This chapter experiments with the common-practice baseline and the lower-order (improved) rest costs. For brevity, the use of lower-order rest costs from Chapter 6 is denoted by “Rest” in plot legends and table captions.

7.4.2 Measurement

CPU time is the sum of user and system time as reported by the kernel. Loading time, defined as the time taken by the decoder on empty input, is subtracted. The French–English system uses sentence-filtered grammars; the time to load these grammars is also counted as part of the CPU time reported. The constant cost of parsing is included in CPU time. Time is averaged over all sentences in the test set.

All experiments were run on an otherwise-idle identical machines with 64 GB RAM and 32 cores, the same machines used in Chapter 3 for toolkit comparison and in all of Chapter 6. No experiment came close to running out of memory. In all cases, files were copied to local disk then forced into the operating system (Linux) disk cache before starting the clock.

Decoders were run single-threaded[2] and compiled for speed as suggested in their documentation. Moses (Hoang et al., 2009) used tcmalloc (Ghemawat, 2011), which increases the speed of memory allocation. The in-memory rule table is used with Moses because it is faster than the on-disk option. Benchmarks are based on Moses revision 26d3940 (with an updated version of this work) and cdec (Dyer et al., 2010) revision 8dc383a.

Search accuracy is measured by average model score; higher is better. Only relative comparisons are meaningful because model scores have arbitrary scale and include constant factors. Beam sizes start at 5 and generally rise until a time limit determined by running ordinary cube pruning with beam size 1000. However, the French–English baseline is very slow, so baseline cube pruning was run with beam sizes up to 300.

The goal of this chapter is not to increase translation quality, but to attain the same quality faster. Nonetheless, it is important to verify that translation quality is not somehow degraded by this work. Translation quality is measured approximately with uncased BLEU (Papineni et al., 2002) and METEOR 1.4 (Denkowski and Lavie, 2011) tuned for HTER (Snover et al., 2006). The relationship between model score and automatic metric scores is generally noisy and differences are generally small. The Chinese–English system has four references while the other systems have one reference.

Feature weights for all systems were tuned toward BLEU using baseline cube pruning. They are unchanged from the original translation systems used in evaluations. This ensures that scores are comparable and eliminates a potential source of noise.

7.4.3 Comparison with Cube Pruning

Cube pruning is the primary baseline. Section 7.2.2 explained that there are multiple variants of cube pruning. Moses implements both normal cube pruning and additive cube pruning (the difference being that the latter delays calling the language model until after a hypothesis is popped from the priority queue). Both are shown as baselines when Moses is run. Gesmundo and Henderson (2010) have implemented their two algorithms in cdec. These appear in the results as “Gesmundo 1” and “Gesmundo 2”; the algorithm numbers correspond to their paper.

Time-Accuracy

The primary result is that, with the exception of beam sizes below 20 for target-syntax systems, this work improves upon the time-accuracy trade-off presented by cube pruning. Figure 7.8 shows the trade-offs for the hierarchical German–English system in Moses. Detailed results for all seven scenarios appear in Appendix A, specifically Figures A.1 through A.7 and Tables A.1 through A.7. This work, indicated by blue squares, is closer to the top left corner (higher accuracy and lower CPU time) of each plot.

Table 7.2 summarizes the results by reporting speed ratios: baseline CPU time divided by this work’s CPU time. Comparisons are made by taking every baseline data point and finding a data point from this work with equal or better accuracy, resulting in a range of speed ratios. The values were then minimized over baselines, thereby reflecting the strongest baseline. For example, cube pruning on Chinese–English with a beam size of 1000 and baseline rest costs attained a model score of -82.332 in 2.41 seconds per sentence. With rest costs and a beam size of 150, this work attained the slightly better model score of -82.330 in 0.32 second per sentence. The speed ratio is

$$\frac{2.41}{0.32} = 7.5$$

[2]Improvement is also seen in multi-threaded Moses, but cdec only supports one thread. Running single-threaded also reduces noise in timing measurements.

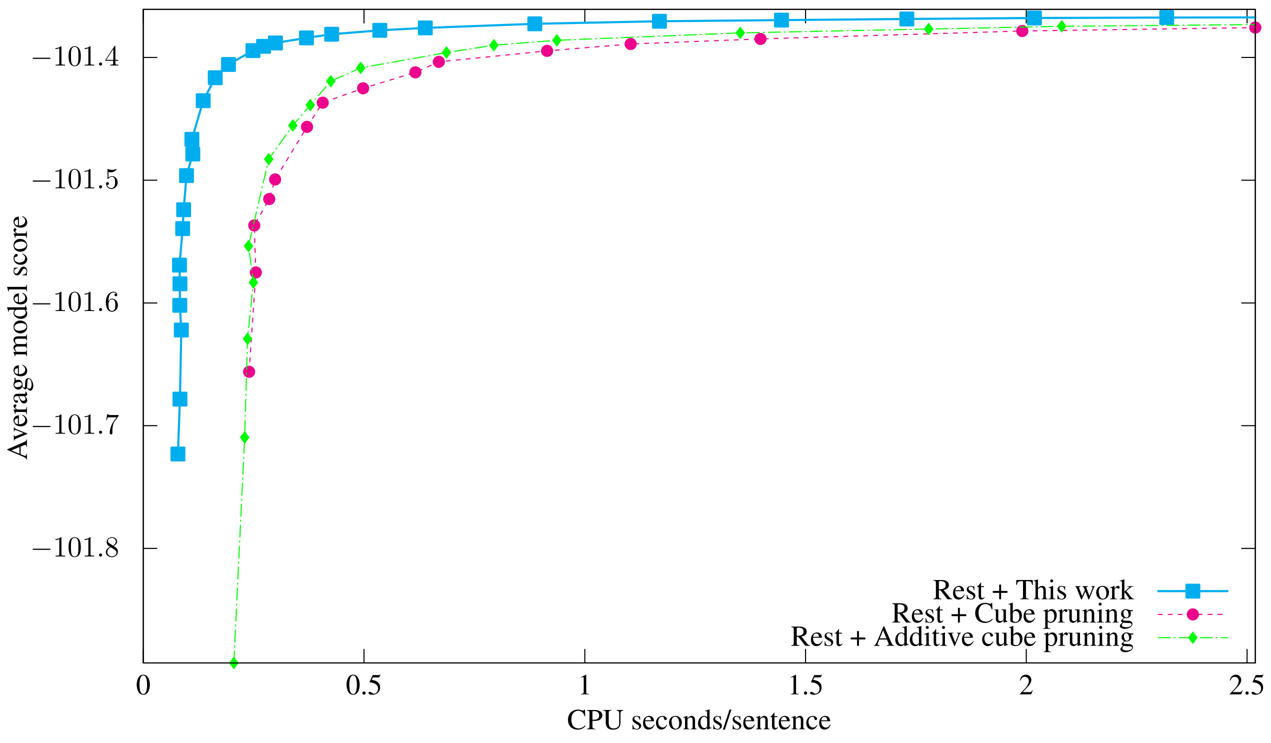


Figure 7.8: Time-accuracy trade-off for the hierarchical German–English system in Moses.

System	Decoder	Baseline Rest Costs All Beams	Baseline Rest Costs Beam $\geq$ 20	Improved Rest Costs All Beams	Improved Rest Costs Beam $\geq$ 20	Rest+This vs Baseline All Beams	Rest+This vs Baseline Beam $\geq$ 20
hierarchical zh–en	Moses	2.15–2.77	2.15–2.77	2.12–3.41	2.12–3.41	1.97–6.60	2.12–6.60
hierarchical en–de	Moses	1.79–2.74	1.79–2.74	1.99–2.94	1.99–2.83	2.24–3.34	2.24–3.34
hierarchical de–en	Moses	2.01–2.97	2.01–2.86	2.35–3.04	2.35–3.04	2.50–4.68	2.50–4.68
hierarchical de–en	cdec	1.60–2.04	1.60–2.04	1.62–2.53	1.62–2.53	1.47–3.90	1.69–3.90
target syntax en–de	Moses	0.49–1.71	1.19–1.71	0.60–1.88	1.27–1.88	0.88–2.13	1.60–2.13
target syntax de–en	Moses	0.77–1.88	1.15–1.75	0.86–2.08	1.17–2.08	1.15–4.17	1.87–4.17
tree-to-tree fr–en	cdec	1.43–2.94	1.46–2.94	1.41–4.15	1.57–4.15	1.47–4.50	1.62–4.50

Table 7.2: The primary result: speed ratios relative to the best cube pruning variant for each system. Comparisons are shown for all algorithms with baseline rest costs, all algorithms with lower order rest costs from Chapter 6, and the net gain in speed by stacking lower order rest costs from Chapter 6 with this chapter. The second column in each scenario excludes baseline data points with beam sizes below 20.

System	Decoder	Baseline Rest Costs All Beams	Baseline Rest Costs Beam $\geq$ 20	Improved Rest Costs All Beams	Improved Rest Costs Beam $\geq$ 20	Rest+This vs Baseline All Beams	Rest+This vs Baseline Beam $\geq$ 20
hierarchical de–en	cdec	2.04–3.08	2.04–2.77	2.23–3.34	2.23–3.16	2.19–5.18	2.55–5.18
tree-to-tree fr–en	cdec	2.78–4.04	2.78–4.03	3.55–6.37	3.61–6.37	4.55–8.23	4.55–8.23

Table 7.3: Speed ratios similar to Table 7.2 except that parsing time has been subtracted before computing the ratios.

but the last column of Table 7.2 reports a smaller value because additive cube pruning is the stronger baseline in this case.

Generally, the largest improvement is seen on hierarchical systems while smaller (and, for beam sizes below 20, negative) improvement is seen on target syntax systems. This is an instance of a more general phenomenon: when there are many hypotheses in a hypergraph vertex, this work performs well because there are more opportunities to group hypotheses together. When there are few hypotheses, the root node branches into several distinct hypotheses and the algorithm reduces to additive cube pruning. In Moses (Hoang et al., 2009), hypergraph vertices covering the same span share the same beam limit, even if there are multiple labels. Thus, in the presence of multiple labels, each hypergraph vertex generally contains fewer hypotheses. Moreover, there may be popular and unpopular labels, so even large beam sizes exhibit a mixture of small and large numbers of hypotheses in vertices. In cdec (Dyer et al., 2010), each hypergraph vertex is allowed the full beam size, explaining why the tree-to-tree system (which has even more labels) is less impacted by this issue.

Rest costs from Chapter 6 improve the performance of both this work and the baselines. The impact on this work is generally larger, as indicated by larger ratios in the "Improved Rest Costs" columns of Table 7.2.

The hierarchical German–English system was decoded with both Moses and cdec (details appear in Figures A.3 and A.4). Speed ratios are generally lower with cdec. The reason is that, as noted in Section 2.4.3, cdec is slower at parsing, so ratios are smoothed towards one. Moreover, cdec's cube pruning implementation is slighly more accurate for low beam sizes, perhaps because it always tries the top-scoring target-side rule for each source-side rule.

Because cdec parses before searching, modifying the decoder to skip search is straightforward. Running the modified decoder produces no useful output, but timing the decoder indicates how long parsing takes. Parsing time is 0.198 second per sentence in the hierarchical German–English system and 4.685 seconds per sentence in the French–English system (including sentence-filtered grammar loading). Table 7.3 shows speed ratios after these times are subtracted. Because Moses searches on the fly during parsing and search can influence parsing (by not producing a hypothesis with a non-terminal that exists in the parse), it is harder to separate the costs, and no such value is reported for Moses.

Performance by Beam Size

While this work generally has better performance in terms of the time-accuracy trade-off, one might wonder whether these results are due to improved accuracy or increased speed. Results by beam size for the hierarchical German–English system are shown in Figures 7.9 and 7.10; detailed results appear in Figures A.8 through A.14 and Tables A.1 through A.7.

As noted in the previous section, this work is generally less accurate for small beam sizes. However, it quickly takes the lead in accuracy for all hierarchical systems, such as the German–English system shown in Figure 7.9. For syntactic systems, this work is less accurate, though improved rest costs narrow the gap (e.g. Figure A.14 in the appendix). One possible explanation is that grouping biases the search algorithm towards finding similar hypotheses instead of diverse hypotheses. Similar hypotheses have only a few grammar labels, so there are fewer chances that the grammar will find a hypothesis it likes.

This work is faster across the board in terms of speed, both in terms of constant and variable costs. An example is shown in Figure 7.10. Speed ranges from 1.19 times as fast as Moses additive cube pruning in hierarchical Chinese–English (Figure A.8) with beam size 750 to 3.90 times as fast as cdec's Gesmundo and Henderson (2010) Algorithm 1 on the tree-to-tree French–English system (Figure A.14) with beam size 300. As seen in the previous section, these speed improvements more than compensate for reduced accuracy on syntactic systems.

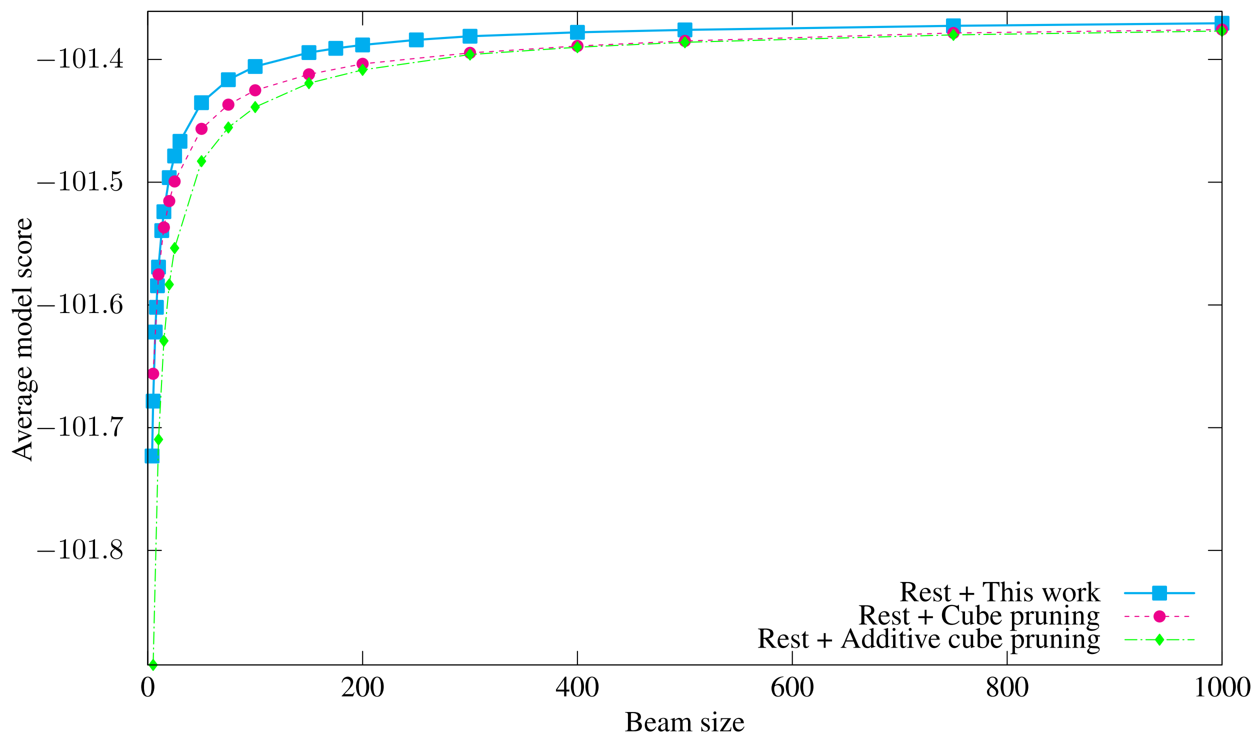


Figure 7.9: Model score as a function of beam size for the hierarchical German–English system in Moses.

Translation Quality

This work makes translation faster but neither improves nor degrades translation quality. An example trade-off between CPU time and quality, as approximated by BLEU, is shown in Figure 7.11. Full results appear in Figures A.15 through A.21 in the appendix.

Very low beam sizes lead to low quality scores. Beyond this point, the relationship is noisy. Systems were tuned towards BLEU, so the noise in those plots indicates model error. A special case is fortuitious search error, wherein search errors lead to better translation quality. Fortuitious search errors appear in Figure 7.11 and other results as a small spike in BLEU score followed by a decline. Both model error and fortuitious search error are well-known phenomena (Koehn, 2010). Nonetheless, improving search accuracy is still worthwhile both to reach the peak faster and to enable future work on improving translation models.

7.4.4 Comparison with Coarse-to-Fine

Coarse-to-fine decoding (Zhang and Gildea, 2008; Petrov et al., 2008) refers to a wide variety of techniques that perform multiple decoding passes. Each pass refines the model and prunes the search space for the next pass. This section compares with the most related version of coarse-to-fine decoding, where the language model order increases. The algorithm is implemented by cdec (Dyer et al., 2010) but not Moses (Hoang et al., 2009).

There are a number of parameters to choose: the orders of language models to use, the amount of pruning to perform in each pass, and the beam size for cube pruning. Rather than test a limited number of parameters with a large number of systems, several options are tested with one system: hierarchical German–English. Baseline rest costs are used, though coarse-to-fine decoding is provided with e.g. an actual bigram language

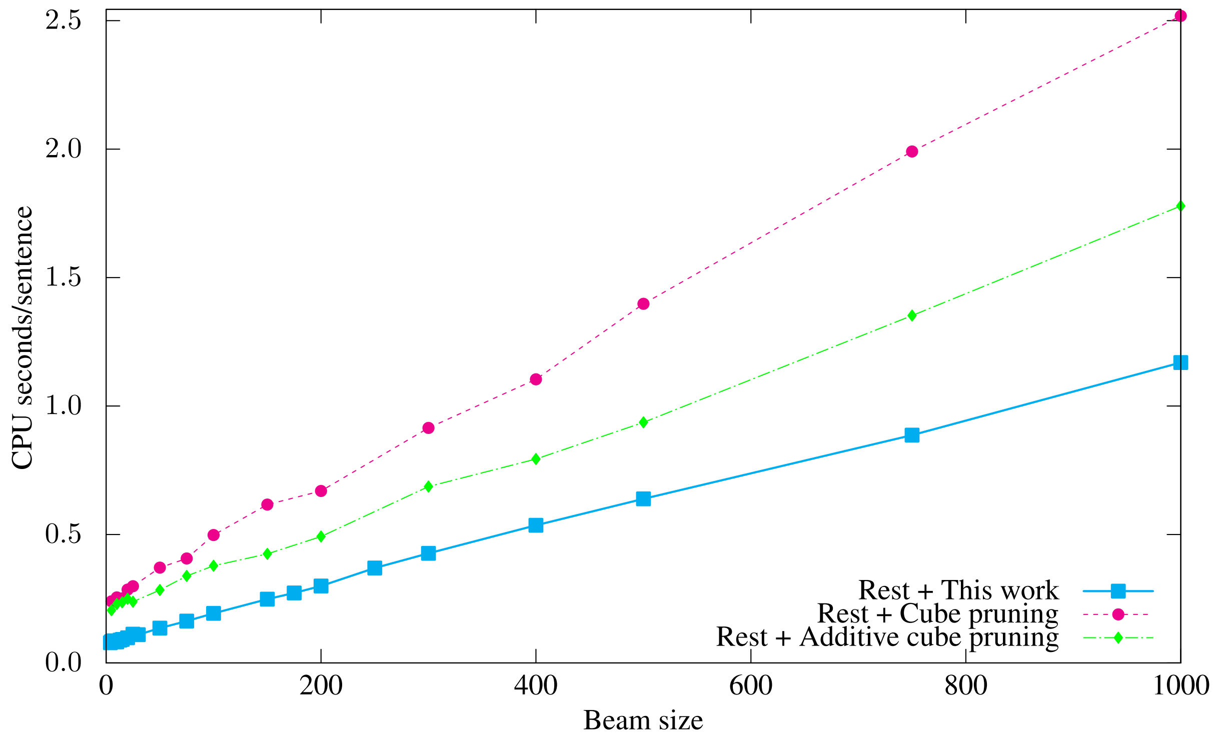


Figure 7.10: CPU time as a function of beam size for the hierarchical German–English system in Moses.

model for a bigram pass instead of simply using lower-order entries from the 5–gram language model. Each pass of coarse-to-fine decoding performs cube pruning.[3] Experiments use the cube pruning variant that performed best on this task, Algorithm 1 from Gesmundo and Henderson (2010).

Each pass except the last performs hypergraph pruning with cdec's density pruning option. This option preserves a fixed (user-specified) number of hypotheses in each hypergraph node. Counting is performed after language model recombination. Language model recombination is based on Chapter 5 for the language model order used in the decoding pass.

Zhang and Gildea (2008) suggest an initial pass with a bigram language model. Figure 7.12 shows results with a bigram pass followed by a full 5–gram pass and density pruning settings 100, 150, and 300. According to the speed ratio defined in Section 7.4.3, this work is 1.41–2.80 times as fast as coarse-to-fine decoding with an initial bigram pass. Table 7.2 reported a speed ratio of 1.60–1.98 relative to cube pruning baselines. The time-accuracy trade-off improves for low beam sizes, but degrades for higher beam sizes, where bigram pruning has removed useful hypotheses. The density pruning setting could be increased further in an effort to increase accuracy for higher beam sizes, but doing so slows the decoder down. For example, with beam size 500 and density pruning 300, coarse-to-fine decoding is actually slower than baseline cube pruning.

Given that the bigram pass prunes good hypotheses or increases CPU time, it makes less sense to use it as the first in a series of refinements. Higher accuracy is attained with a trigram pass followed by a 5–gram pass (and no bigram pass). Figure 7.12 shows the results. The differences are relatively small compared to noise. Nonetheless, there is a pattern: density pruning to 50 works best for small beam sizes but loses to the baseline on larger beam sizes, 100 works best in the middle, and 300 wins on larger beam sizes. Computing the speed ratios defined in Section 7.4.3, this work is 1.40–1.77 times as fast as coarse-to-fine decoding with an initial trigram pass, compared to a ratio of 1.60–1.98 relative to cube pruning.

[3]In theory, this work could be used instead of cube pruning, but the implementation currently does not support it.

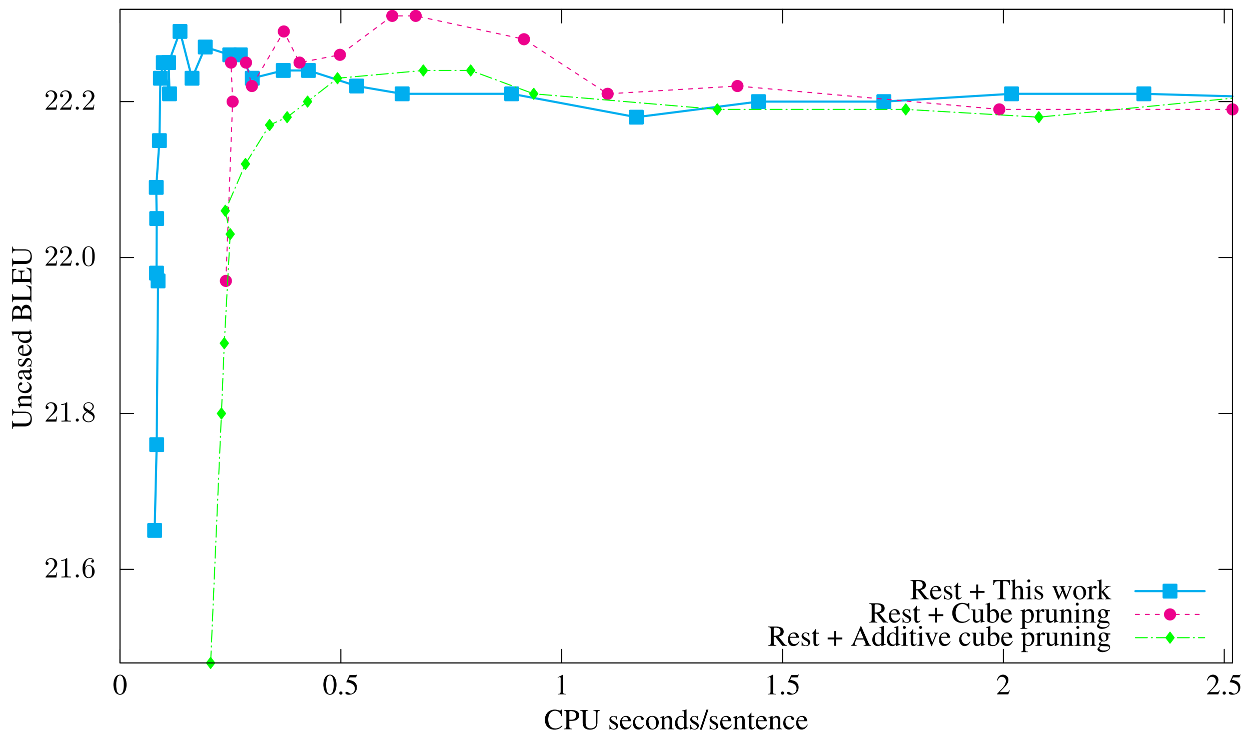


Figure 7.11: Trade-off between CPU time and BLEU score in the hierarchical German–English system with Moses. These results include lower-order rest costs from Chapter 6.

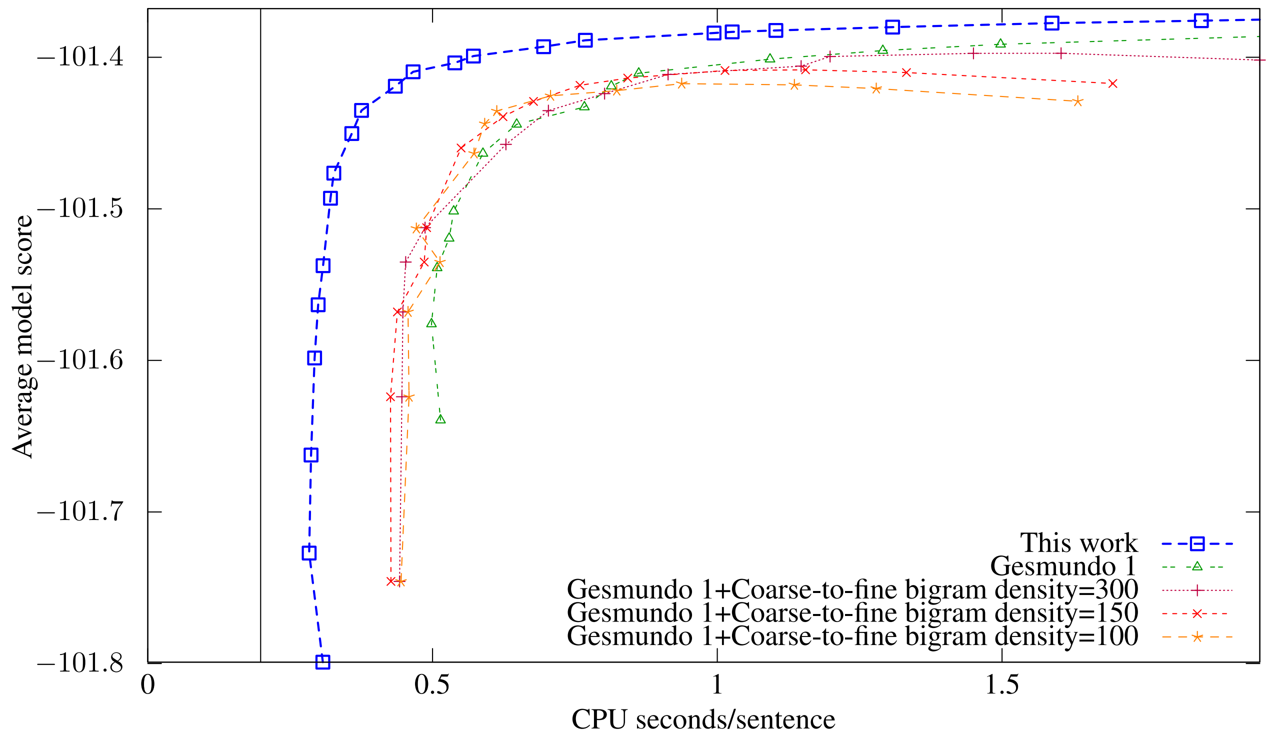
7.4.5 Revealing Only Left or Right Words

The state trees alternate between revealing words on the left and on the right of a partial state. In some cases, words on one side of state do not matter, so revealing them introduces spurious ambiguity. Specifically, if the target-side grammar rule begins with a non-terminal, then the left state of that non-terminal does not matter for purposes of selecting the top k beam entries. This case can be optimized with a state tree that reveals only words from right state. Similarly, non-terminals at the end of a grammar rule can be optimized with a state tree that reveals only words from left state.

Figure 7.13 shows results for the German–English hierarchical and target syntax systems, both with improved rest costs. There is essentially no change in performance. When a spurious word is revealed, the score of the partial edge does not change, so the partial edge will still be at the top of the priority queue and visited in the next iteration. Thus, this optimization mostly impacts speed by reducing roundtrips to the priority queue and handling more hypotheses in the same group. However, that speedup is counterbalanced by the cost of maintaining three separate state trees: one for left state only, one for right state only, and one that alternates. One might expect to have seen improvement from glue rule applications $S \rightarrow SX$ by only revealing right state of S and left state of X. However, S has empty left state because glued hypotheses are built from left to right, so the only difference is that X does not reveal right state.

7.4.6 Memory

Peak virtual memory usage was measured before each process terminated. In the hierarchical German–English system with a beam size of 1000, this work uses 160 MB more RAM in Moses and 298 MB less



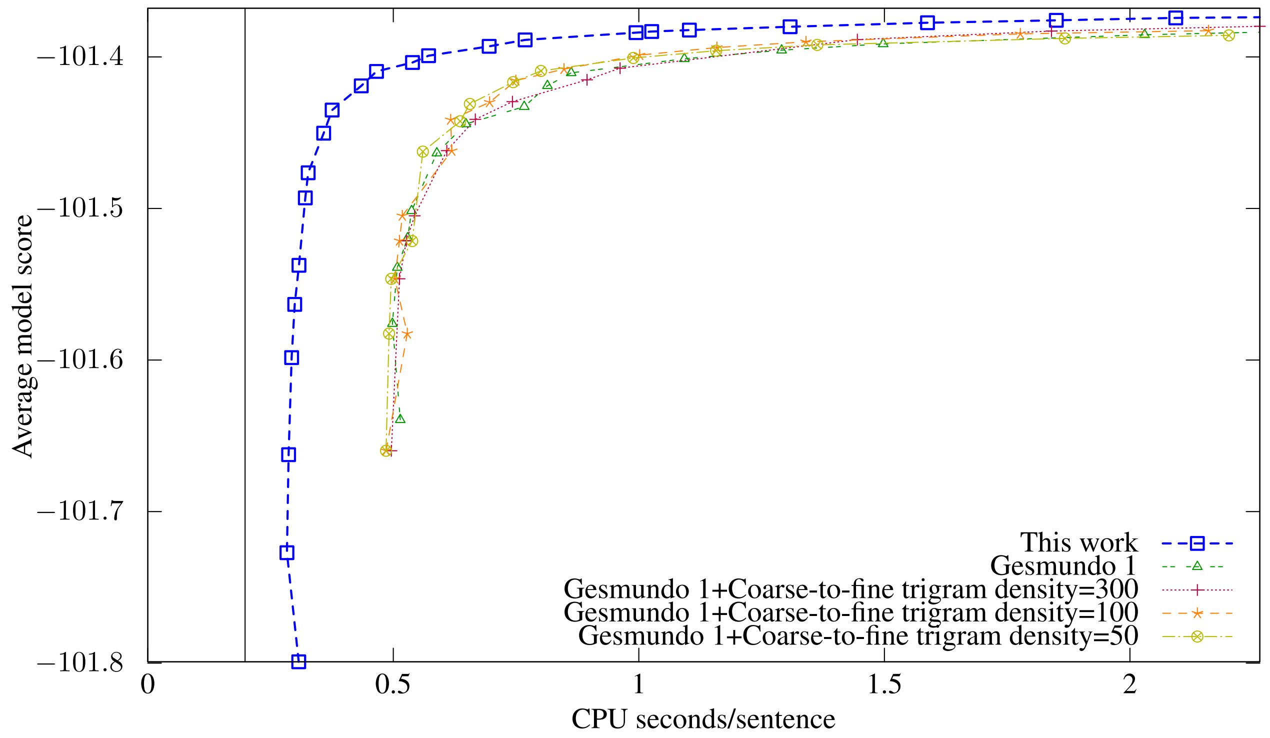
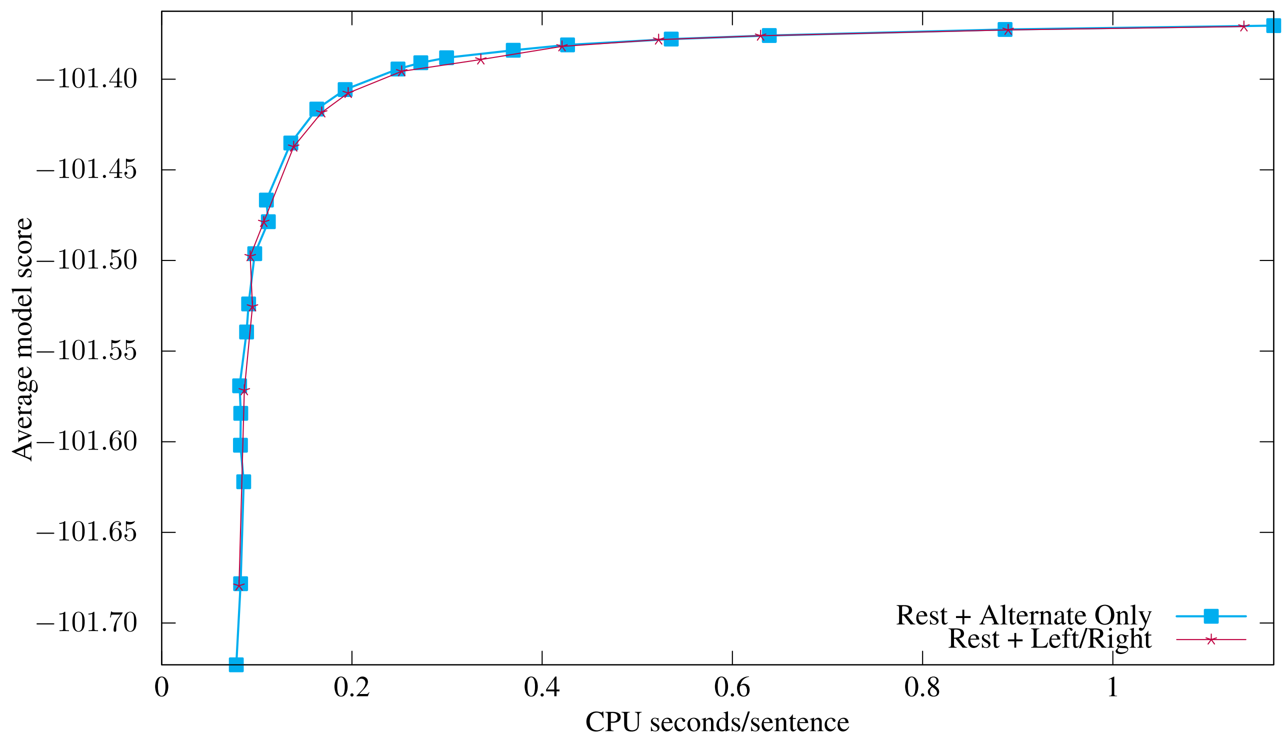


Figure 7.12: Coarse-to-fine decoding in **cdec on hierarchical German–English** with bigram and 5–gram passes (top) or trigram and 5–gram passes (bottom).



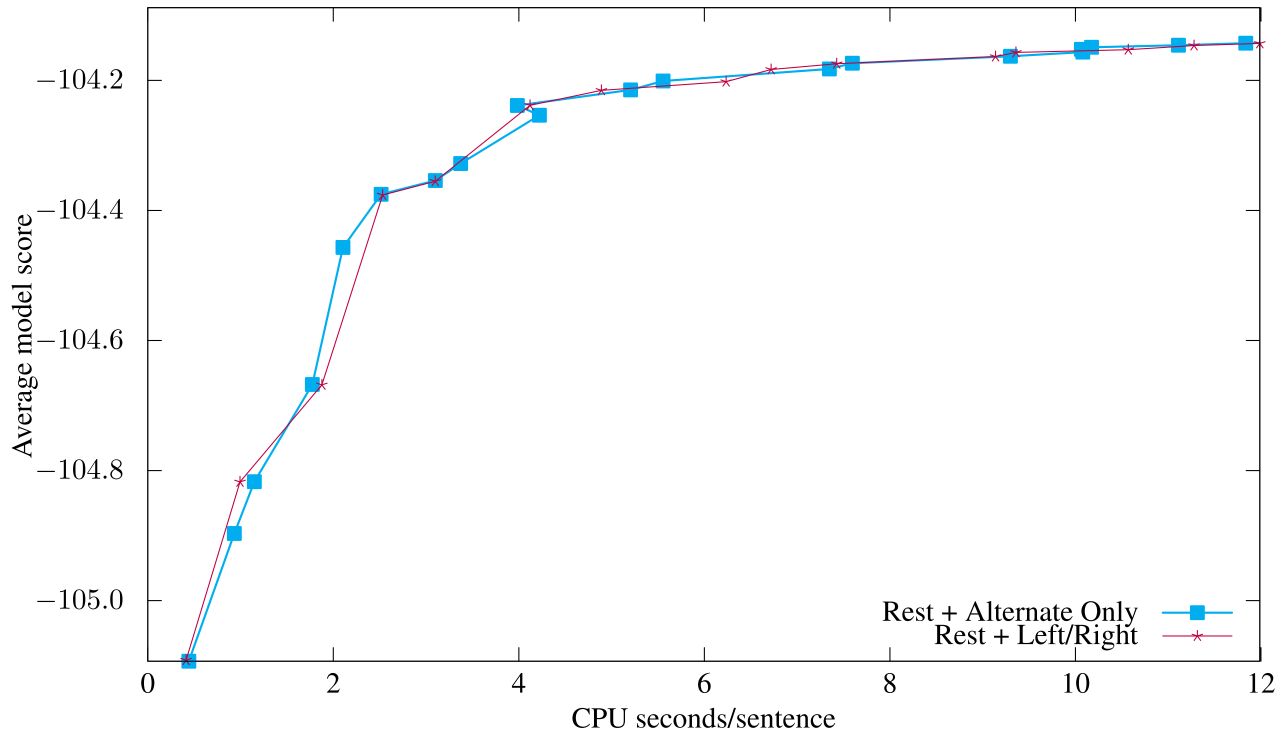


Figure 7.13: Special handling of left and right state (Section 7.4.5) for non-terminals at the beginning and end of a rule had little impact on performance of **Moses on the German–English** hierarchical (top) and target syntax systems (bottom).

RAM in cdec compared to ordinary cube pruning. The differences generally small (especially with lower beam sizes) and minor relative to 12-13 GB total size of the process, most of which is the phrase table and language model.

7.5 Summary

This chapter described a new search algorithm that achieves equivalent accuracy 1.15–6.60 times as fast as several cube pruning variants, including coarse-to-fine decoding, for beam sizes at least 20. Improvement has been demonstrated across six language pairs and three grammar formalisms. The algorithm is based on grouping similar language model feature states together and dynamically expanding these groups. In doing so, it exploits the language model's ability to estimate with incomplete information. The algorithm builds on efficient querying from Chapter 4, state handling from Chapter 5, and (optionally) the improved rest costs from Chapter 6.

Chapter 8

Conclusions and Future Work

If you talk to a man in a language he understands, that goes to his head. If you talk to him in his language, that goes to his heart. –Nelson Mandela

In making a speech one must study three points: first, the means of producing persuasion; second, the language; third the proper arrangement of the various parts of the speech. –Aristotle

This thesis contributes efficient algorithms for N-gram language modeling. Performance gains from these algorithms are useful to exploit large corpora, speed experimentation, and move research from supercomputers to production (Numen, 2012; FEAT Limited, 2012). Contributions cover every step where language models are used in a statistical system: estimating from text, querying for probabilities, exploiting independence assumptions, scoring with incomplete information, and searching for high-scoring hypotheses.

General-purpose language modeling benefits from disk-based streaming algorithms that estimate unpruned modified interpolated Kneser-Ney language models. Because the user determines how much RAM to use, dramatically larger language models can be built on a single machine. Moreover, one might expect disk-based algorithms to be slow but, surprisingly, it is faster in terms of wall time than all toolkits tested.

Many applications depend on fast language model queries. Local RAM is the fastest, but available memory can be tight. This work contributes two data structures, along with optional variants, that improve the trade-off between query speed and memory consumption. This is accomplished with custom hash tables and interpolation search. Further compression can be obtained by collapsing probability and backoff into a single value, while provably maintaining sentence-level probabilities.

Log language model probability is a common feature in many statistical systems. In addition to efficient language model representation, these systems can benefit from encoding independence assumptions that increase recombination, carrying useful information from one query to the next to increase speed, better rest costs in the presence of incomplete information that increase speed and/or search accuracy, and a new search approximate algorithm. The new search algorithm exploits the language model's tendency to treat hypotheses similarly if they share common prefixes or suffixes.

Syntactic machine translation was the primary motivation for much of the work on decoding. Nonetheless, most of the resulting work, including the new search algorithm, still applies to lattice decoding problems that arise in other applications. Numen (2012) have applied contributions from every chapter, except Chapter 6 (Rest Costs), to lattice decoding for optical character recognition. Their language model could also be compressed as described in Chapter 6, but is currently not large enough to require such compression. Others have applied Chapter 4 (Queries) to efficiently query language models in automatic speech recognition systems (Kim et al., 2012; Si et al., 2013).

Code, dubbed KenLM (most chapters) and lazy (search), is available under the LGPL[1] and already in use by the community.

8.1 Future Work

8.1.1 Estimating Language Models

Many have requested that the disk-based language model estimation algorithm be extended to prune n-grams with low adjusted counts. Support for pruning is not as trivial as removing entries. The substring property of Section 5.3.3 should be maintained in a scalable way rather than deleting and later hallucinating approximate probabilities for some entries. The join performed in Section 3.3.5 currently assumes that every n-gram is the context of an $(n + 1)$-gram unless it ends with the unknown word or the end of sentence marker; this property does not hold in the presence of pruning.

Interpolating language models trained on separate data is a common practice.[2] Currently, researchers are dependent on SRILM (Stolcke, 2002) to produce interpolated language models, effectively enforcing the same size limitations as before. The same streaming and sorting infrastructure could be used to statically[3] interpolate language models.

Kneser-Ney smoothing (Kneser and Ney, 1995) is not well-defined if there is no singleton. This can happen for part of speech language models, where smoothing is nonetheless useful for higher orders. Other smoothing algorithms such as Witten-Bell (Witten and Bell, 1991), Good-Turing (Good, 1953), and Jelinek-Mercer (Jelinek and Mercer, 1980) might be scalably implemented using the same sorting and streaming infrastructure.

The present work uses threads for different tasks, such as block sorting. This is less effective than using threads to process pieces of the data in parallel. Brants et al. (2007) have shown how the set of n-grams can be partitioned for parallel processing in some cases. Extending this work to partition data across threads should dramatically decrease wall time. More generally, a cluster version of the streaming and sorting framework would generalize MapReduce (Dean and Ghemawat, 2004) and is reminiscent of Dryad (Isard et al., 2007).

8.1.2 Language Model Data Structures

Applications such as first- or single-pass speech recognition expect the language model to recommend continuations given some history. This differs from machine translation and lattice decoding, where the words to query are given by the search space. Neither the large hash table nor the reverse trie is laid out in a way that makes recommendations efficient. Other data structures, such as a forward trie implemented by IRSTLM (Federico et al., 2008), are more efficient at recommending continuations but less efficient at performing queries.

Minimal perfect hashing (Talbot and Brants, 2008; Guthrie and Hepple, 2010; Belazzougui et al., 2008) is effective at lossily compressing language models. Prior work has largely applied minimal perfect hashing to store models smoothed with stupid backoff (Brants et al., 2007), in part because estimating Kneser-Ney models was too expensive before Chapter 3. Language models in the Katz (1987) family, including interpolated modified Kneser-Ney (Kneser and Ney, 1995; Chen and Goodman, 1998) could be stored in the same memory by applying Chapter 6 to collapse probability and backoff into a single value then quantizing

[1] `http://kheafield.com/code/`

[2] This practice is orthogonal to interpolating orders within the same language model, which is supported by the present implementation.

[3] Static interpolation builds a single model with interpolated probabilities. The alternative is dynamic interpolation, in which all models are loaded into RAM, as implemented by IRSTLM (Federico et al., 2008).

(Whittaker and Raj, 2001; Federico and Bertoldi, 2006). Doing so should substantially reduce the memory size of the 643 GB language model used in Section 3.6.4.

8.1.3 Rest Costs

Currently, the improved rest costs from Chapter 6 are stored as an additional floating-point value alongside probability and backoff. Since the rest cost is close to the conditional probability, the rest cost could be encoded by quantizing the difference between the rest cost and the conditional probability. The number of bits used to quantize should trade between search accuracy (more bits) and memory consumption (fewer bits), effectively creating a continuum between the two options. Sentence-level probabilities would not change because only rest costs are quantized.

Carter et al. (2012) defined an upper bound on language model probability by taking the maximum over all possible histories. However, the only requirement for upper bounds is that they multiply to at least the probability of the sentence. By applying the same telescoping series idea that made it possible to collapse probability and backoff into a single value, it is possible to shift some responsbility from upper($w_n \mid w_1^{n-1}$ to upper($w^{n-1} \mid w_1^{n-2}$). Thus, histories would become partly responsible for the upper bound on probability of the next word. It may be possible to tune this responsibility shifting in a way that optimizes exact search algorithms such as Aziz et al. (2013).

8.1.4 Search Algorithms

Beam search is not ideal: the same number of hypotheses are extracted in each vertex without regard to their diversity or usefulness to subsequent vertices. There has been some effort on varying beam sizes (Xu et al., 2013), but doing so actually made search less accurate. Much more generally, revealing words and visiting vertices could be interwoven. As they do in the present work, parents would decide which boundary words from children are worth exploring and which are not, but the children would fill these requests lazily instead of basing them on k hypotheses. Moreover, the children need not generate complete hypotheses to fill the requests, but rather recursively request just enough boundary words from their children in order to do so. In other words, the algorithm uses a mixture of recursive top-down demand and recursive bottom-up response to search for high-scoring hypotheses. Open questions lie in the areas of where to prioritize refinement, whether parents are agreessively or lazily informed of updates to dynamic state trees in their children, and in memory-efficient ways to keep track of parts of the space that have already been explored. In some sense, the proposed algorithm is a mixture of this work and the exact decoding algorithm of Aziz et al. (2013). The key differences from Aziz et al. (2013) are that parsing is performed once, partially evaluated hypotheses are dynamically updated instead of discarding and reconstructing them, updates are performed locally instead of every place where an n-gram appears, and the score need not be an upper bound on language model probability (so search can be exact or approximate).

Class-based language models have been reported to work with coarse-to-fine decoding (Petrov et al., 2008). The search algorithm could be extended to reveal the class of a boundary word before it reveals the actual word, thereby increasing opportunities to group hypotheses. Moreover, the search algorithm implementation could be extended to make it a compatible subroutine of coarse-to-fine decoding, just as cube pruning is currently used as a subroutine.

Bibliography

James M. Abello and Jeffrey Scott Vitter, editors. 1999. *External memory algorithms*. American Mathematical Society, Boston, MA, USA.

Alfred V. Aho and Jeffrey D. Ullman. 1969. Syntax directed translations and the pushdown assembler. *Journal of Computer and System Sciences*, 3:37–56, February.

Alexander Allauzen, Nicolas Pécheux, Quoc Khanh Do, Marco Dinarelli, Thomas Lavergne, Aurélien Max, Hai-Son Le, and François Yvon. 2013. LIMSI @ WMT13. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 62–69, Sofia, Bulgaria, August. Association for Computational Linguistics.

Alexander Allauzen. 2013. Personal communication, August.

Waleed Ammar, Victor Chahuneau, Michael Denkowski, Greg Hanneman, Wang Ling, Austin Matthews, Kenton Murray, Nicola Segall, Yulia Tsvetkov, Alon Lavie, and Chris Dyer. 2013. The CMU machine translation systems at WMT 2013: Syntax, synthetic translation options, and pseudo-references. In *Proceedings of the ACL 2013 Eighth Workshop on Statistical Machine Translation*, pages 70–77, Sofia, Bulgaria, August.

Austin Appleby. 2012. SMHasher & MurmurHash. `https://code.google.com/p/smhasher/`.

Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. 2013. Nobody ever got fired for buying a cluster. Technical Report MSR-TR-2013-2, Microsoft Research.

Wilker Aziz, Marc Dymetman, and Sriram Venkatapathy. 2013. Investigations in exact inference for hierarchical translation. In *Proceedings of the ACL 2013 Eighth Workshop on Statistical Machine Translation*, Sofia, Bulgaria, August.

Lalit R. Bahl, Peter F. Brown, Peter V. de Souza, and Robert L. Mercer. 1989. A tree-based statistical language model for natural language speech recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(7):1001–1008.

Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. 1964. *On Formal Properties of Simple Phrase Structure Grammars*. Hebrew University Students' Press.

Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. 2008. Hash, displace, and compress. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming (ICALP '08)*, pages 385–396.

Nicola Bertoldi. 2013. Personal communication.

Ergun Biciçi. 2013. Feature decay algorithms for fast deployment of accurate statistical machine translation systems. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 78–84, Sofia, Bulgaria, August. Association for Computational Linguistics.

Karel Bílek and Daniel Zeman. 2013. CUni multilingual matrix in the WMT 2013 shared task. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 85–91, Sofia, Bulgaria, August. Association for Computational Linguistics.

Alexandra Birch, Phil Blunsom, and Miles Osborne. 2009. A quantitative analysis of reordering phenomena. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*, pages 197–205, Athens, Greece, March. Association for Computational Linguistics.

Dina Bitton and David J DeWitt. 1983. Duplicate record elimination in large data files. *ACM Transactions on database systems (TODS)*, 8(2):255–265.

Burton Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July.

Ondřej Bojar, Christian Buck, Chris Callison-Burch, Christian Federmann, Barry Haddow, Philipp Koehn, Christof Monz, Matt Post, Radu Soricut, and Lucia Specia. 2013a. Findings of the 2013 Workshop on Statistical Machine Translation. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 1–44, Sofia, Bulgaria, August. Association for Computational Linguistics.

Ondřej Bojar, Christian Buck, Chris Callison-Burch, Barry Haddow, Philipp Koehn, Christof Monz, Matt Post, Herve Saint-Amand, Radu Soricut, and Lucia Specia, editors. 2013b. *Proceedings of the Eighth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, Sofia, Bulgaria, August.

Ondřej Bojar, Rudolf Rosa, and Aleš Tamchyna. 2013c. Chimera – three heads for English-to-Czech translation. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 92–98, Sofia, Bulgaria, August. Association for Computational Linguistics.

Ondřej Bojar. 2013. Personal communication, July.

Alexey Borisov, Jacob Dlougach, and Irina Galinskaya. 2013. Yandex school of data analysis machine translation systems for WMT13. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 99–103, Sofia, Bulgaria, August. Association for Computational Linguistics.

Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Language Learning*, pages 858–867, June.

Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. 1984. *Classification and Regression Trees*. Chapman and Hall/CRC.

Jamie Callan, Mark Hoy, Changkuk Yoo, and Le Zhao. 2009. The ClueWeb09 dataset. `http://lemurproject.org/clueweb09/`.

Chris Callison-Burch, Philipp Koehn, Christof Monz, and Omar Zaidan. 2011. Findings of the 2011 workshop on statistical machine translation. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 22–64, Edinburgh, Scotland, July. Association for Computational Linguistics.

Chris Callison-Burch, Philipp Koehn, Christof Monz, Matt Post, Radu Soricut, and Lucia Specia. 2012. Findings of the 2012 workshop on statistical machine translation. In *Proceedings of the Seventh Workshop on Statistical Machine Translation*, pages 10–51, Montréal, Canada, June. Association for Computational Linguistics.

Simon Carter, Marc Dymetman, and Guillaume Bouchard. 2012. Exact sampling and decoding in high-order hidden Markov models. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1125–1134, Jeju Island, Korea, July.

Jean-Cédric Chappelier, Martin Rajman, Ramón Aragüés, Antoine Rozenknop, et al. 1999. Lattice parsing for speech recognition. In *Proc. of 6ème conférence sur le Traitement Automatique du Langage Naturel (TALN 99)*, pages 95–104.

Ciprian Chelba and Johan Schalkwyk, 2013. *Empirical Exploration of Language Modeling for the google.com Query Stream as Applied to Mobile Voice Search*, pages 197–229. Springer, New York.

Ciprian Chelba, Thorsten Brants, Will Neveitt, and Peng Xu. 2010. Study on interaction between entropy pruning and Kneser-Ney smoothing. In *Proceedings of Interspeech*, pages 2242–2245.

Stanley Chen and Joshua Goodman. 1998. An empirical study of smoothing techniques for language modeling. Technical Report TR-10-98, Harvard University, August.

Siyuan Chen, Dharitri Misra, and George R. Thoma. 2010. Efficient automatic OCR word validation using word partial format derivation and language model. In *Document Recognition and Retrieval XVII: Proceedings of the SPIE*, volume 7534.

Colin Cherry and George Foster. 2012. Batch tuning strategies for statistical machine translation. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 427–436. Association for Computational Linguistics.

David Chiang. 2005. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 263–270, Ann Arbor, Michigan, June.

David Chiang. 2007. Hierarchical phrase-based translation. *Computational Linguistics*, 33:201–228, June.

Eunah Cho, Thanh-Le Ha, Mohammed Mediani, Jan Niehues, Teresa Herrmann, Isabel Slawik, and Alex Waibel. 2013. The Karlsruhe Institute of Technology translation systems for the WMT 2013. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 104–108, Sofia, Bulgaria, August. Association for Computational Linguistics.

Miranda Chong and Lucia Specia. 2012. Linguistic and statistical traits characterising plagiarism. In *Proceedings of the 24th International Conference on Computational Linguistics*, pages 195–204, Mumbai, India.

Jonathan Chow. 2012. Personal communication, May.

Tagyoung Chung and Michel Galley. 2012. Direct error rate minimization for statistical machine translation. In *Proceedings of the Seventh Workshop on Statistical Machine Translation*, pages 468–479, Montréal, Canada, June. Association for Computational Linguistics.

Kenneth Church, Ted Hart, and Jianfeng Gao. 2007. Compressing trigram language models with Golomb coding. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 199–207, Prague, Czech Republic, June. Association for Computational Linguistics.

Philip Clarkson and Ronald Rosenfeld. 1997. Statistical language modeling using the CMU-Cambridge toolkit. In *Proceedings of Eurospeech*.

Michael Collins. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania.

Gordon V Cormack, Mark D Smucker, and Charles LA Clarke. 2011. Efficient and effective spam filtering and re-ranking for large web datasets. *Information retrieval*, 14(5):441–465.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, USA, 12.

Roman Dementiev, Lutz Kettner, and Peter Sanders. 2008. STXXL: standard template library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637.

Kris Demuynck, Dirk Van Compernolle, and Patrick Wambacq. 2002. Doing away with the Viterbi approximation. In *Proceedings of the 2002 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 1. IEEE.

Michael Denkowski and Alon Lavie. 2011. Meteor 1.3: Automatic metric for reliable optimization and evaluation of machine translation systems. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 85–91, Edinburgh, Scotland, July. Association for Computational Linguistics.

Michael Denkowski. 2013. Machine translation for human translators. Ph.D. Thesis Proposal, Carnegie Mellon University, May.

Richard O. Duda and Peter E. Hart. 1973. *Pattern Classification and Scene Analysis*. Wiley, New York, NY.

Ilknur Durgar El-Kahlout and Coşkun Mermer. 2013. TÜbİtak-bİlgem german-english machine translation systems for w13. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 109–113, Sofia, Bulgaria, August. Association for Computational Linguistics.

Nadir Durrani, Alexander Fraser, and Helmut Schmid. 2013a. Model with minimal translation units, but decode with phrases. In *Proceedings of NAACL-HLT*, pages 1–11.

Nadir Durrani, Alexander Fraser, Helmut Schmid, Hassan Sajjad, and Richárd Farkas. 2013b. Munich-Edinburgh-Stuttgart submissions of OSM systems at WMT13. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 122–127, Sofia, Bulgaria, August. Association for Computational Linguistics.

Nadir Durrani, Barry Haddow, Kenneth Heafield, and Philipp Koehn. 2013c. Edinburgh's machine translation systems for European language pairs. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 114–121, Sofia, Bulgaria, August. Association for Computational Linguistics.

Nadir Durrani, Barry Haddow, Kenneth Heafield, and Philipp Koehn. 2013d. Edinburgh's machine translation systems for European language pairs. In *Proceedings of the ACL 2013 Eighth Workshop on Statistical Machine Translation*, Sofia, Bulgaria, August.

Nadir Durrani. 2013. Personal communication, July.

Chris Dyer, Adam Lopez, Juri Ganitkevitch, Johnathan Weese, Ferhan Ture, Phil Blunsom, Hendra Setiawan, Vladimir Eidelman, and Philip Resnik. 2010. cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models. In *Proceedings of the ACL 2010 System Demonstrations*, ACLDemos '10, pages 7–12.

Chris Dyer. 2013. Personal communication, June.

Vladimir Eidelman, Ke Wu, Ferhan Ture, Philip Resnik, and Jimmy Lin. 2013. Towards efficient large-scale feature-rich statistical machine translation. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 128–133, Sofia, Bulgaria, August. Association for Computational Linguistics.

FEAT Limited. 2012. Voicetra+. `http://voicetra-plus.jp/voicetra.html`, December.

Marcello Federico and Nicola Bertoldi. 2006. How many bits are needed to store probabilities for phrase-based translation? In *Proceedings of the Workshop on Statistical Machine Translation*, pages 94–101, New York City, June.

Marcello Federico, Nicola Bertoldi, and Mauro Cettolo. 2008. IRSTLM: an open source toolkit for handling large scale language models. In *Proceedings of Interspeech*, Brisbane, Australia.

Christian Federmann. 2007. Very large language models for machine translation. Master's thesis, Saarland University.

Lluis Formiga, Carlos A. Henríquez Q., Adolfo Hernández, José B. Mariño, Enric Monte, and José A. R. Fonollosa. 2012. The TALP-UPC phrase-based translation systems for wmt12: Morphology simplification and domain adaptation. In *Proceedings of the Seventh Workshop on Statistical Machine Translation*, pages 275–282, Montréal, Canada, June. Association for Computational Linguistics.

Lluís Formiga, Marta R. Costa-jussà, José B. Mariño, José A. R. Fonollosa, Alberto Barrón-Cedeño, and Lluis Marquez. 2013. The TALP-UPC phrase-based translation systems for WMT13: System combination with morphology generation, domain adaptation and corpus filtering. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 134–140, Sofia, Bulgaria, August. Association for Computational Linguistics.

Michel Galley and Christopher D. Manning. 2008. A simple and effective hierarchical phrase reordering model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 848–856, Honolulu, Hawaii. Association for Computational Linguistics.

Petra Galuščáková, Martin Popel, and Ondřej Bojar. 2013. PhraseFix: Statistical post-editing of TectoMT. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 141–147, Sofia, Bulgaria, August. Association for Computational Linguistics.

Jianfeng Gao and Jonathan H. Clark. 2013. Personal communication, April.

Ulrich Germann, Eric Joanis, and Samuel Larkin. 2009. Tightly packed tries: How to fit large models into memory, and make them load fast, too. In *Proceedings of the NAACL HLT Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 31–39, Boulder, Colorado.

Ulrich Germann. 2011. Speed of tightly packed tries. Personal communication, May.

Andrea Gesmundo and James Henderson. 2010. Faster cube pruning. In *Proceedings of the International Workshop on Spoken Language Translation (IWSLT)*, pages 267–274.

Sanjay Ghemawat. 2011. TCMalloc: Thread-caching malloc, July. `https://gperftools.googlecode.com/svn/trunk/doc/tcmalloc.html`.

Irving Good. 1953. The population frequencies of species and the estimation of population parameters. *Biometrika*, 40, December.

Spence Green, Daniel Cer, Kevin Reschke, Rob Voigt, John Bauer, Sida Wang, Natalia Silveira, Julia Neidert, and Christopher D. Manning. 2013. Feature-rich phrase-based translation: Stanford University's submission to the WMT 2013 translation task. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 148–153, Sofia, Bulgaria, August. Association for Computational Linguistics.

David Guthrie and Mark Hepple. 2010. Storing the web in memory: Space efficient language models with constant time retrieval. In *Proceedings of EMNLP 2010*, Los Angeles, CA.

Greg Hanneman and Alon Lavie. 2013. Improving syntax-augmented machine translation by coarsening the label set. In *Proceedings of NAACL-HLT 2013*, Atlanta, Georgia. Association for Computational Linguistics.

Peter Hart, Nils Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July.

Eva Hasler, Peter Bell, Arnab Ghoshal, Barry Haddow, Philipp Koehn, Fergus McInnes, Steve Renals, and Pawel Swietojanski. 2012. The UEDIN systems for the IWSLT 2012 evaluation. In *Proceedings of IWSLT 2012*, Hong Kong, December.

Kenneth Heafield and Alon Lavie. 2010. Combining machine translation output with open source: The Carnegie Mellon multi-engine machine translation scheme. *The Prague Bulletin of Mathematical Linguistics*, 93:27–36.

Kenneth Heafield, Hieu Hoang, Philipp Koehn, Tetsuo Kiso, and Marcello Federico. 2011. Left language model state for syntactic machine translation. In *Proceedings of the International Workshop on Spoken Language Translation*, San Francisco, CA, USA, December.

Kenneth Heafield, Philipp Koehn, and Alon Lavie. 2012. Language model rest costs and space-efficient storage. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, Jeju Island, Korea.

Kenneth Heafield, Philipp Koehn, and Alon Lavie. 2013a. Grouping language model boundary words to speed k-best extraction from hypergraphs. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Atlanta, Georgia, USA, June.

Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H. Clark, and Philipp Koehn. 2013b. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, Sofia, Bulgaria, August.

Kenneth Heafield. 2011. KenLM: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, Edinburgh, UK, July. Association for Computational Linguistics.

Hieu Hoang, Philipp Koehn, and Adam Lopez. 2009. A unified framework for phrase-based, hierarchical, and syntax-based statistical machine translation. In *Proceedings of the International Workshop on Spoken Language Translation*, pages 152–159, Tokyo, Japan.

Hieu Hoang. 2013. Personal communication, July.

Mark Hopkins and Greg Langmead. 2009. Cube pruning as heuristic search. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 62–71, Singapore, August.

Mark Hopkins and Jonathan May. 2011. Tuning as ranking. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1352—-1362, Edinburgh, Scotland, July.

Bo-June Hsu and James Glass. 2008. Iterative language model estimation: Efficient data structure & algorithms. In *Proceedings of Interspeech*, Brisbane, Australia.

Liang Huang and David Chiang. 2007. Forest rescoring: Faster decoding with integrated language models. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics*, Prague, Czech Republic.

Liang Huang and Haitao Mi. 2010. Efficient incremental decoding for tree-to-string translation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 273–283, Cambridge, MA, October.

Songfang Huang and Steve Renals. 2010. Power law discounting for n-gram language models. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'10)*, pages 5178–5181, Dallas, Texas, USA, March.

Zhiheng Huang, Yi Chang, Bo Long, Jean-Francois Crespo, Anlei Dong, Sathiya Keerthi, and Su-Lin Wu. 2012. Iterative Viterbi A* algorithm for k-best sequential decoding. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1125–1134, Jeju Island, Korea, July.

Stéphane Huet, Elena Manishina, and Fabrice Lefèvre. 2013. Factored machine translation systems for Russian-English. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 154–157, Sofia, Bulgaria, August. Association for Computational Linguistics.

Gonzalo Iglesias, Cyril Allauzen, William Byrne, Adrià de Gispert, and Michael Riley. 2011. Hierarchical phrase-based translation representations. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1373–1383, Edinburgh, Scotland, UK, July. Association for Computational Linguistics.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72.

Daniel James. 2005. Boost.Unordered. `http://www.boost.org/doc/libs/release/doc/html/unordered.html`.

Frederick Jelinek and Robert L. Mercer. 1980. Interpolated estimation of markov sourceparameters from sparse data. In *In Proceedings of the Workshop on Pattern Recognition in Practice*, pages 381–397, May.

Frederick Jelinek, Robert L. Mercer, L.R. Bahl, and J.K. Baker. 1977. Perplexity–a measure of the difficult7 of speech recognition tasks. *The Journal of the Acoustical Society of America*, 62.

Frederick Jelinek. 1997. *Statistical Methods for Speech Recognition*. MIT Press, Cambridge, Massachusetts.

Slava Katz. 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(3):400–401, March.

Mark D. Kernighan, Kenneth W. Church, and William A. Gale. 1990. A spelling correction program based on a noisy channel model. In *Proceedings of the 13th conference on computational linguistics*, pages 205–210, Helsinki, Finland, August.

Jungsuk Kim, Jike Chong, and Ian Lane. 2012. Efficient on-the-fly hypothesis rescoring in a hybrid GPU/CPU-based large vocabulary continuous speech recognition engine. In *Proceedings of the 13th Annual Conference of the International Speech Communication Association*, Portland, Oregon, September.

Dan Klein and Christopher D. Manning. 2001. Parsing and hypergraphs. In *Proceedings of the Seventh International Workshop on Parsing Technologies*, Beijing, China, October.

Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 181–184.

Kevin Knight. 1999. Decoding complexity in word-replacement translation models. *Computational Linguistics*, 25:607–615.

Donald E. Knuth. 1963. Notes on "open" addressing. Unpublished memorandum, July.

Philipp Koehn and Barry Haddow. 2012. Towards effective use of training data in statistical machine translation. In *Proceedings of the Seventh Workshop on Statistical Machine Translation*, pages 317–321, Montréal, Canada, June. Association for Computational Linguistics.

Philipp Koehn, Franz Josef Och, and Daniel Marcu. 2003. Statistical phrase-based translation. In *Proceedings of HLT-NAACL 2003*, pages 48–54, Edmonton, Canada.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open source toolkit for statistical machine translation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, Prague, Czech Republic, June.

Philipp Koehn. 2005. Europarl: A parallel corpus for statistical machine translation. In *Proceedings of MT Summit*.

Philipp Koehn. 2010. *Statistical Machine Translation*. Cambridge University Press.

Philipp Koehn. 2011. Personal communication, September.

Okan Kolak, William Byrne, and Philip Resnik. 2003. A generative probabilistic OCR model for NLP applications. In *Proceedings of HLT-NAACL 2003*, Edmonton, Canada.

Shankar Kumar and William Byrne. 2004. Minimum Bayes-risk decoding for statistical machine translation. In *Proceedings of the Human Language Technology conference / North American chapter of the Association for Computational Linguistics annual meeting*, Boston, MA.

William Lewis and Sauleh Eetemadi. 2013. Dramatically reducing training data size through vocabulary saturation. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 281–291, Sofia, Bulgaria, August. Association for Computational Linguistics.

Zhifei Li and Sanjeev Khudanpur. 2008. A scalable decoder for parsing-based machine translation with equivalent language model state maintenance. In *Proceedings of the Second ACL Workshop on Syntax and Structure in Statistical Translation (SSST-2)*, pages 10–18, Columbus, Ohio, June.

Zhifei Li, Chris Callison-Burch, Chris Dyer, Sanjeev Khudanpur, Lane Schwartz, Wren Thornton, Jonathan Weese, and Omar Zaidan. 2009. 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. Association for Computational Linguistics.

Bruce T. Lowerre. 1976. *The Harpy speech recognition system*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

Evgeny Matusov and Gregor Leusch. 2013. Omnifluent English-to-French and Russian-to-English systems for the 2013 Workshop on Statistical Machine Translation. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 158–163, Sofia, Bulgaria, August. Association for Computational Linguistics.

Eric Mays, Fred Damerau, and Robert Mercer. 1991. Context based spelling correction. *Information Processing & Management*, 27(5):517–522.

Antonio Valerio Miceli Barone and Giuseppe Attardi. 2013. Pre-reordering for machine translation using transition-based walks on dependency parse trees. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 164–169, Sofia, Bulgaria, August. Association for Computational Linguistics.

Antonio Valerio Miceli Barone. 2013. Personal communication, August.

Tomas Mikolov, Martin Karafiát, Lukáš Burget, Jan Cernocky, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Proceedings of Interspeech*, pages 1045–1048.

Behrang Mohit, Rebecca Hwa, and Alon Lavie. 2010. Using variable decoding weight for language model in statistical machine translation. In *Proceedings of the Ninth Conference of the Association for Machine Translation in the Americas (AMTA-10)*.

Robert Moore and William Lewis. 2010. Intelligent selection of language model training data. In *Proceedings of the ACL 2010 Conference Short Papers*, pages 220–224, Uppsala, Sweden, July. Association for Computational Linguistics.

Robert C. Moore and Chris Quirk. 2009. Less is more: Significance-based n-gram selection for smaller, better language models. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 746–755, August.

Robert Moore, Douglas Appelt, John Dowding, J. Mark Gawron, and Douglas Moran. 1995. Combining linguistic and statistical knowledge sources in natural-language processing for ATIS. In *Procedings of the ARPA Spoken Language Technology Workshop*.

Maria Nadejde, Philip Williams, and Philipp Koehn. 2013. Edinburgh's syntax-based machine translation systems. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 170–176, Sofia, Bulgaria, August. Association for Computational Linguistics.

Patrick Nguyen, Jianfeng Gao, and Milind Mahajan. 2007. MSRLM: a scalable language modeling toolkit. Technical Report MSR-TR-2007-144, Microsoft Research.

Jan Niehues. 2013. Personal communication, August.

Numen. 2012. Project to improve text capture. `http://www.numen.fr/en/innovation-rd/project-improve-text-capture`.

Franz Josef Och and Hermann Ney. 2002. 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. Association for Computational Linguistics.

Franz Josef Och. 2003. Minimum error rate training in statistical machine translation. In *ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*, pages 160–167, Morristown, NJ, USA. Association for Computational Linguistics.

Tsuyoshi Okita, Qun Liu, and Josef van Genabith. 2013. Shallow semantically-informed PBSMT and HPBSMT. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 177–184, Sofia, Bulgaria, August. Association for Computational Linguistics.

Miles Osborne. 2013. Personal communication, July.

Kishore A. Papineni, Salim Roukos, and R. T. Ward. 1997. Feature-based language understanding. In *Proceedings of the European Conference on Speech Communication and Technology*, pages 1435–1438, Rhodes, Greece, September.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evalution of machine translation. In *Proceedings 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, PA, July.

Robert Parker, David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. 2009. English gigaword fourth edition. LDC2009T13.

Robert Parker, David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. 2011. English gigaword fifth edition, June. LDC2011T07.

Adam Pauls and Dan Klein. 2011. Faster and smaller n-gram language models. In *Proceedings of ACL*, Portland, Oregon.

Adam Pauls. 2011. Personal communication, May.

Adam Pauls. 2013. Class MakeKneserNeyArpaFromText, June. https://berkeleylm.googlecode.com/svn-history/r529/trunk/doc/edu/berkeley/nlp/lm/io/MakeKneserNeyArpaFromText.html.

Stephan Peitz, Saab Mansour, Matthias Huck, Markus Freitag, Hermann Ney, Eunah Cho, Teresa Herrmann, Mohammed Mediani, Jan Niehues, Alex Waibel, Alexander Allauzen, Quoc Khanh Do, Bianka Buschbeck, and Tonio Wandmacher. 2013a. Joint WMT 2013 submission of the QUAERO project. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 185–192, Sofia, Bulgaria, August. Association for Computational Linguistics.

Stephan Peitz, Saab Mansour, Jan-Thorsten Peter, Christoph Schmidt, Joern Wuebker, Matthias Huck, Markus Freitag, and Hermann Ney. 2013b. The RWTH Aachen machine translation system for WMT 2013. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 193–199, Sofia, Bulgaria, August. Association for Computational Linguistics.

Stephan Peitz, Jörn Wübker, and Matthias Huck. 2013c. Personal communication.

Yehoshua Perl, Alon Itai, and Haim Avni. 1978. Interpolation search—a log log N search. *Commun. ACM*, 21:550–553, July.

Slav Petrov, Aria Haghighi, and Dan Klein. 2008. Coarse-to-fine syntactic machine translation using language projections. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 108–116, Honolulu, HI, USA, October.

Juan Pino, Aurelien Waite, Tong Xiao, Adrià de Gispert, Federico Flego, and William Byrne. 2013. The University of Cambridge Russian-English system at WMT13. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 200–205, Sofia, Bulgaria, August. Association for Computational Linguistics.

Jim Pitman and Marc Yor. 1997. The two-parameter poisson-dirichlet distribution derived from a stable subordinator. *The Annals of Probability*, 25(2):855–900.

Jay M. Ponte and W. Bruce Croft. 1998. A language modeling approach to information retrieval. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 275–281, New York, NY, USA. ACM.

Matt Post, Juri Ganitkevitch, Luke Orland, Jonathan Weese, Yuan Cao, and Chris Callison-Burch. 2013. Joshua 5.0: Sparser, better, faster, server. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 206–212, Sofia, Bulgaria, August. Association for Computational Linguistics.

Matthew John Post. 2010. *Syntax-based Language Models for Statistical Machine Translation*. Ph.D. thesis, University of Rochester.

Matt Post. 2013. Personal communication, July.

Bhiksha Raj and Ed Whittaker. 2003. Lossless compression of language model structure and word identifiers. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 388–391.

Ronald Rosenfeld. 2000. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278.

Raphael Rubino, Antonio Toral, Santiago Cortés Vaíllo, Jun Xie, Xiaofeng Wu, Stephen Doherty, and Qun Liu. 2013. The CNGL-DCU-Prompsit translation systems for WMT13. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 213–218, Sofia, Bulgaria, August. Association for Computational Linguistics.

Alexander Rush and Michael Collins. 2011. Exact decoding of syntactic translation models through Lagrangian relaxation. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, pages 72–82, Portland, Oregon, USA, June.

Hassan Sajjad, Svetlana Smekalova, Nadir Durrani, Alexander Fraser, and Helmut Schmid. 2013. QCRI-MES submission at WMT13: Using transliteration mining to improve statistical machine translation. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 219–224, Sofia, Bulgaria, August. Association for Computational Linguistics.

Lane Schwartz. 2012. *An Incremental Syntactic Language Model for Statistical Phrase-based Machine Translation*. Ph.D. thesis, University of Minnesota, Minneapolis, Minnesota, February.

Lane Schwartz. 2013. Personal communication, February.

Holger Schwenk, Anthony Rousseau, and Mohammed Attik. 2012. Large, pruned or continuous space language models on a gpu for statistical machine translation. In *Proceedings of the NAACL-HLT workshop on the Future of Language Modeling for HLT*, pages 11–19.

Claude E Shannon. 1951. Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64.

Yujing Si, Zhen Zhang, Qingqing Zhang, Jielin Pan, and Yonghong Yan. 2013. Discriminative language model with part-of-speech for mandarin large vocabulary continuous speech recognition system. In *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*, pages 970–973, Paris, France, March. Atlantis Press.

Khalil Sima'an. 1996. Computational complexity of probabilistic disambiguation by means of tree-grammars. In *Proceedings of the 16th conference on Computational Linguistics*, volume 2, pages 1175–1180.

Matthew Snover, Bonnie Dorr, Richard Schwartz, Linnea Micciulla, and John Makhoul. 2006. A study of translation edit rate with targeted human annotation. In *Proceedings of the 7th Conference of the Association for Machine Translation in the Americas (AMTA-2006)*, pages 223–231, Cambridge, MA, August.

Andreas Stolcke. 2002. SRILM - an extensible language modeling toolkit. In *Proceedings of the Seventh International Conference on Spoken Language Processing*, pages 901–904.

Sara Stymne, Christian Hardmeier, Jörg Tiedemann, and Joakim Nivre. 2013. Tunable distortion limits and corpus cleaning for SMT. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 225–231, Sofia, Bulgaria, August. Association for Computational Linguistics.

David Talbot and Thorsten Brants. 2008. Randomized language models via perfect hash functions. In *Proceedings of ACL-08: HLT*, pages 505–513.

David Talbot and Miles Osborne. 2007. Randomised language modelling for statistical machine translation. In *Proceedings of ACL*, pages 512–519, Prague, Czech Republic.

Christoph Tillmann. 2004. A unigram orientation model for statistical machine translation. In *Proceedings of HLT-NAACL 2004: Short Papers*, pages 101–104, Boston, Massachusetts.

Xiang Tong and David A. Evans. 1996. A statistical approach to automatic OCR error correction in context. In *Proceedings of the Fourth Workshop on Very Large Corpora*, pages 88–100, Copenhagen, Denmark, April.

Darren Erik Vengroff. 1994. A transparent parallel I/O environment. In *DAGS Symposium on Parallel Computation*, pages 117–134.

David Vilar, Daniel Stein, Matthias Huck, and Hermann Ney. 2010. 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.

Andrew Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260 –269, April.

Taro Watanabe, Hajime Tsukada, and Hideki Isozaki. 2006. Left-to-right target generation for hierarchical phrase-based translation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the ACL*, pages 777—–784, Sydney, Australia, July.

Marion Weller, Max Kisselew, Svetlana Smekalova, Alexander Fraser, Helmut Schmid, Nadir Durrani, Hassan Sajjad, and Richárd Farkas. 2013. Munich-Edinburgh-Stuttgart submissions at WMT13: Morphological and syntactic processing for SMT. In *Proceedings of the Eighth Workshop on Statistical Machine Translation*, pages 232–239, Sofia, Bulgaria, August. Association for Computational Linguistics.

Edward Whittaker and Bhiksha Raj. 2001. Quantization-based language model compression. In *Proceedings of Eurospeech*, pages 33–36, Aalborg, Denmark, December.

Ian Witten and Timothy Bell. 1991. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094.

Deyi Xiong, Min Zhang, and Haizhou Li. 2011. Enhancing language models in statistical machine translation with backward n-grams and mutual information triggers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, pages 1288–1297, Portland, Oregon, June.

Wenduan Xu and Philipp Koehn. 2012. Extending hiero decoding in Moses with cube growing. *The Prague Bulletin of Mathematical Linguistics*, 98:133–142.

Wenduan Xu, Yue Zhang, Philip Williams, and Philipp Koehn. 2013. Learning to prune: Context-sensitive pruning for syntactic MT. In *Proceedings of The 51st Annual Meeting of the Association for Computational Linguistics*, August.

Kenji Yamada and Kevin Knight. 2001. A syntax-based statistical translation model. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 523–530. Association for Computational Linguistics.

Mikhail Zaslavskiy, Marc Dymetman, and Nicola Cancedda. 2009. Phrase-based statistical machine translation as a traveling salesman problem. In *Proceedings of the 47th Annual Meeting of the ACL and the 4th IJCNLP of the AFNLP*, pages 333–341, Suntec, Sinagapore, August.

Daniel Zeman. 2012. Data issues of the multilingual translation matrix. In *Proceedings of the Seventh Workshop on Statistical Machine Translation*, pages 395–400, Montréal, Canada, June. Association for Computational Linguistics.

Richard Zens and Hermann Ney. 2008. Improvements in dynamic programming beam search for phrase-based statistical machine translation. In *Proceedings of the International Workshop on Spoken Language Translation (IWSLT)*, Honolulu, Hawaii, October.

Hao Zhang and Daniel Gildea. 2008. Efficient multi-pass decoding for synchronous context free grammars. In *Proceedings of ACL-08: HLT*, pages 209–217, Columbus, Ohio.

Andreas Zollmann and Ashish Venugopal. 2006. Syntax augmented machine translation via chart parsing. In *Proceedings on the Workshop on Statistical Machine Translation*, pages 138–141, New York City, June. Association for Computational Linguistics.

Appendix A

Search Experiments

This appendix contains detailed experimental results for the improved search algorithm described in Chapter 7. As explained in Section 7.4.1, there are five Moses (Hoang et al., 2009) systems, one tree-to-tree cdec (Dyer et al., 2010) system, and a port of the hierarchical German–English Moses system to cdec, for a total of seven scenarios.

Each scenario was run with baseline rest costs and improved (lower-order) rest costs from Chapter 6. Baseline rest costs appear on the left half of the page with darker colors and hallow plot symbols. Improved rest costs appear on the right half of the page with ligher colors, filled plot symbols, and the word "Rest" in the plot legends and table headings.

In most cases, the top left corner of each plot is best. For example, the new search algorithm generally achieves higher accuracy in less CPU time. The exception is plots that report CPU time as a function of beam size, where the bottom-right corner is best; the new search algorithm consistently uses less CPU time.

As noted in Section 7.4.3, it is possible to measure parsing cost in cdec. The parsing cost appears as a black line whenever cdec is used (Figures A.4, A.11, A.18, A.7, A.14, and A.21).

Search accuracy shows a small jump in the German–English system with target syntax (Figure A.6) around 15 seconds. This is due to translating "durchzogenen" as "criss-crossed" instead of passing it through, which incurs a severe penalty (-100). The only rule capable of doing so translates "X durchzogenen" as "criss-crossed *PP*"; a direct translation rule was not extracted due to reordering. An appropriate prepositional phrase (PP) was pruned with smaller beam sizes and only became available with larger beam sizes.

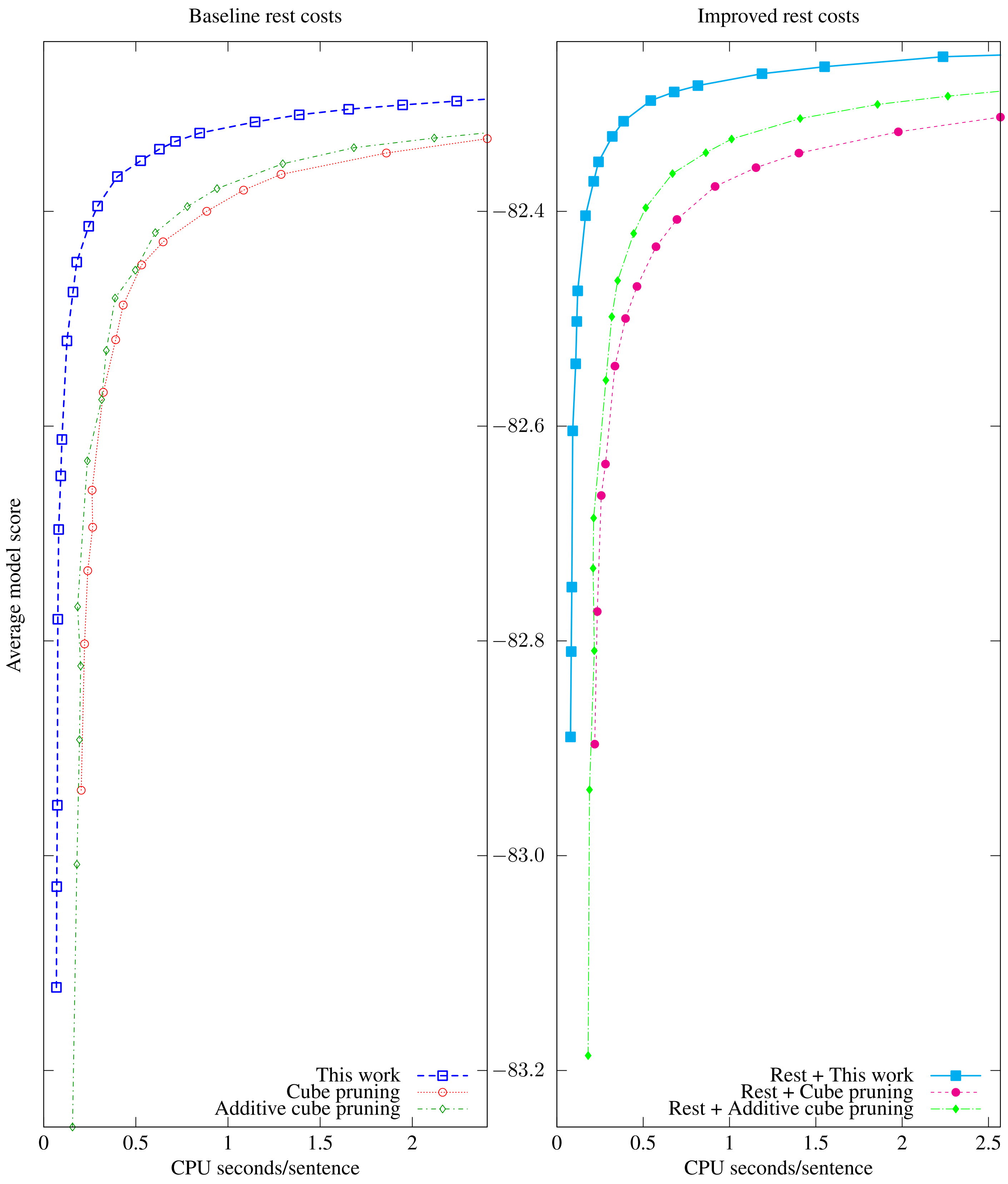


Figure A.1: Time-accuracy trade-off for **Moses on hierarchical Chinese–English**.

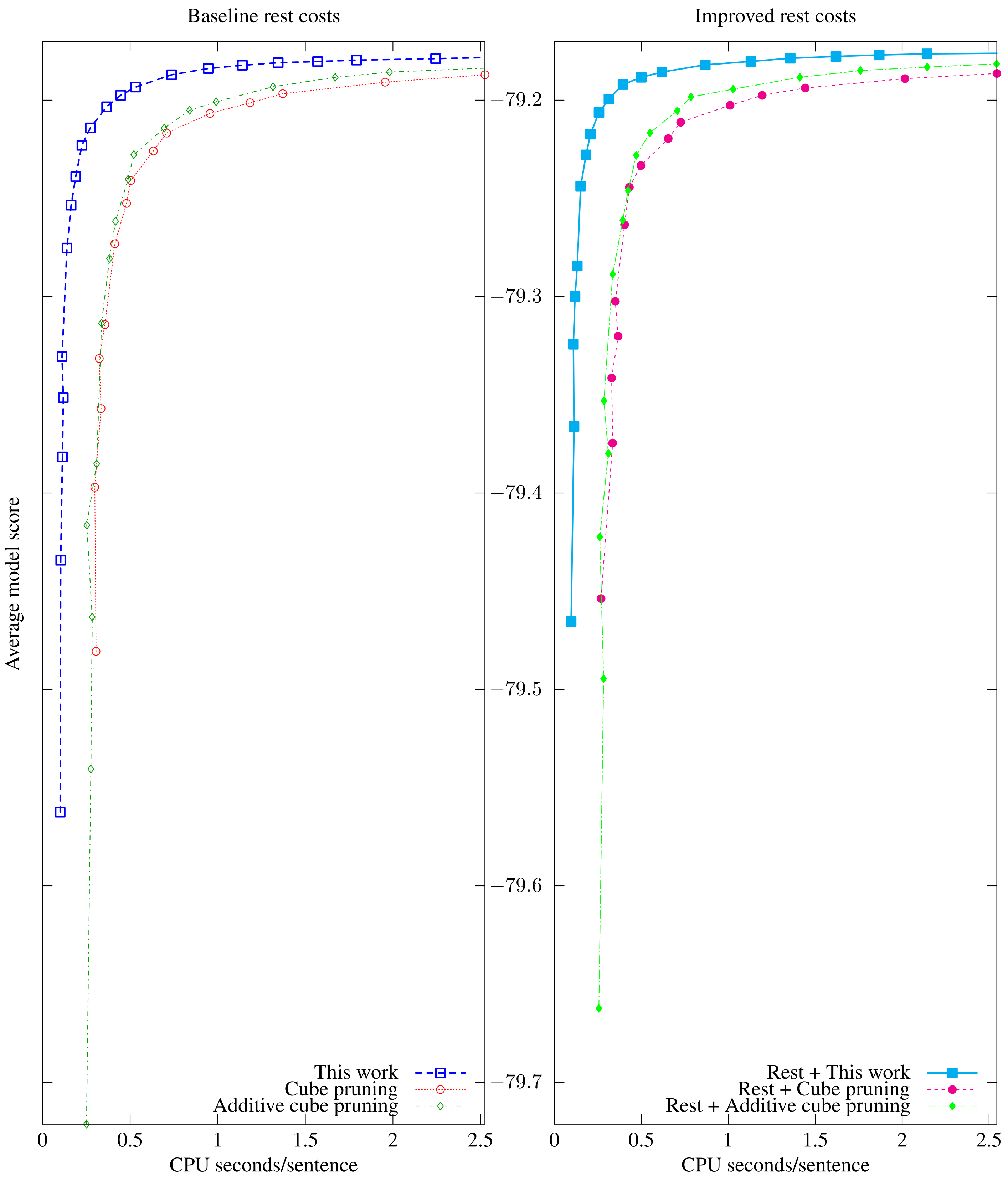


Figure A.2: Time-accuracy trade-off for **Moses on hierarchical English–German**.

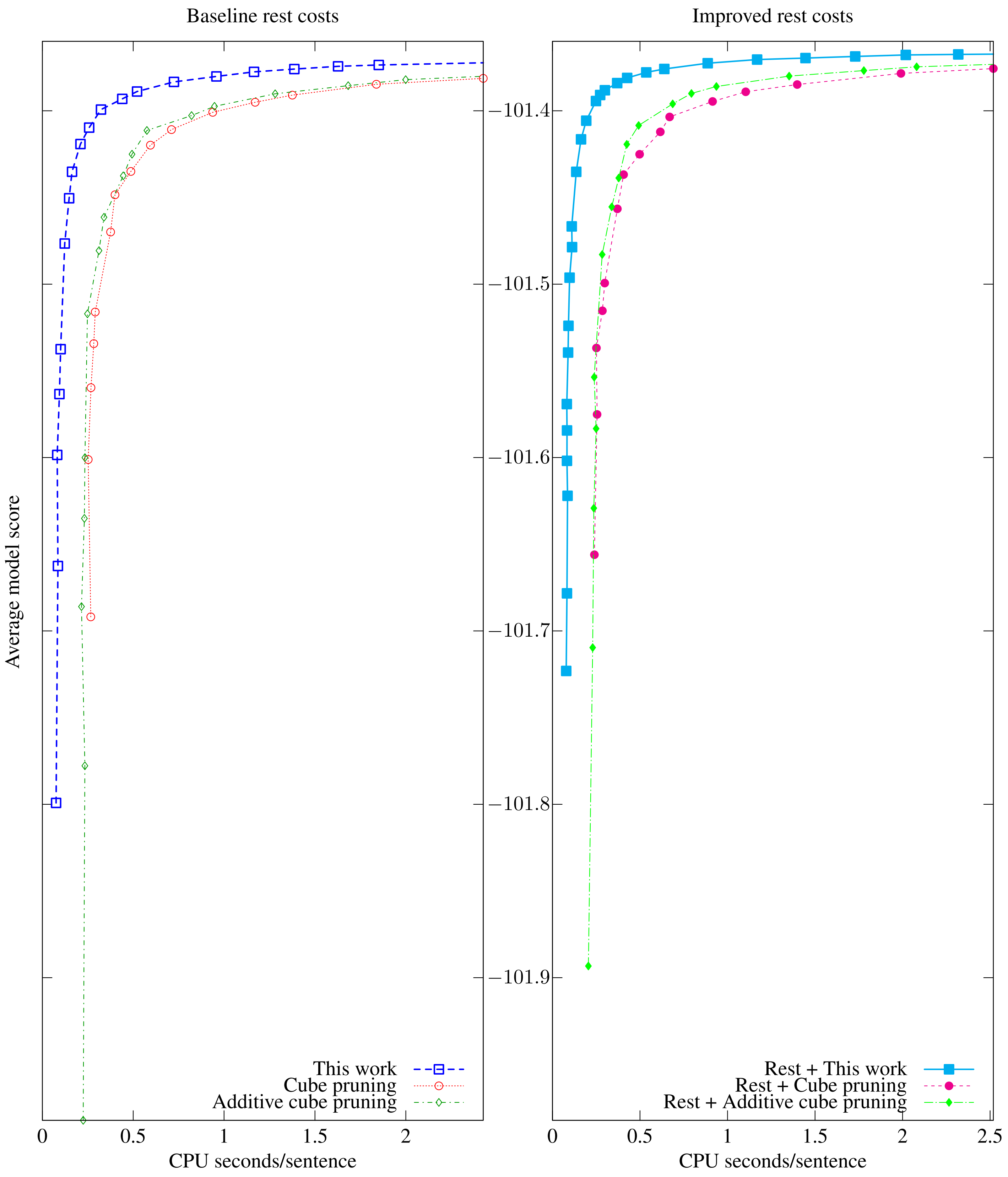


Figure A.3: Time-accuracy trade-off for **Moses on hierarchical German–English**.

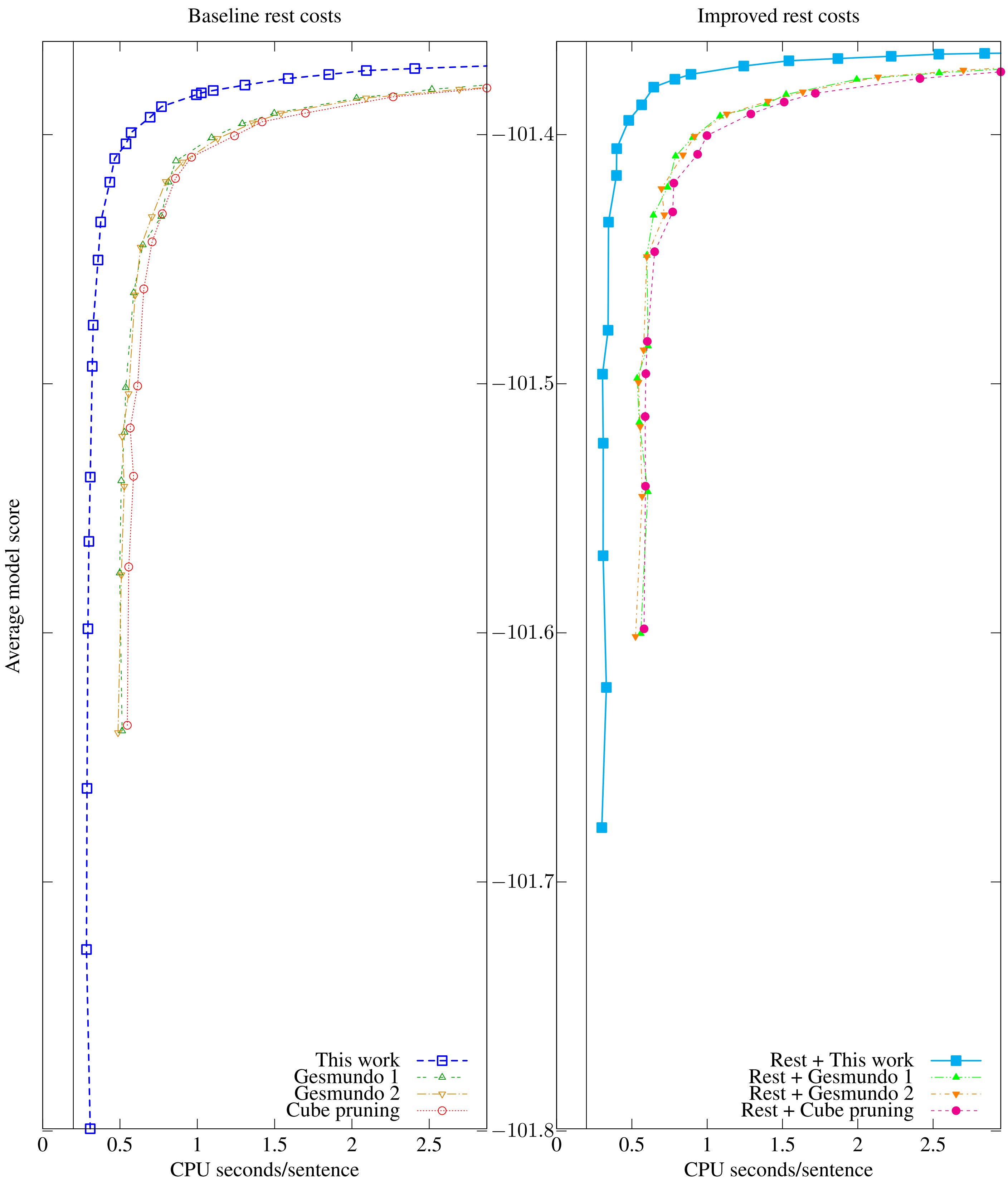


Figure A.4: Time-accuracy trade-off for **cdec on hierarchical German–English**. The black line indicates parsing cost.

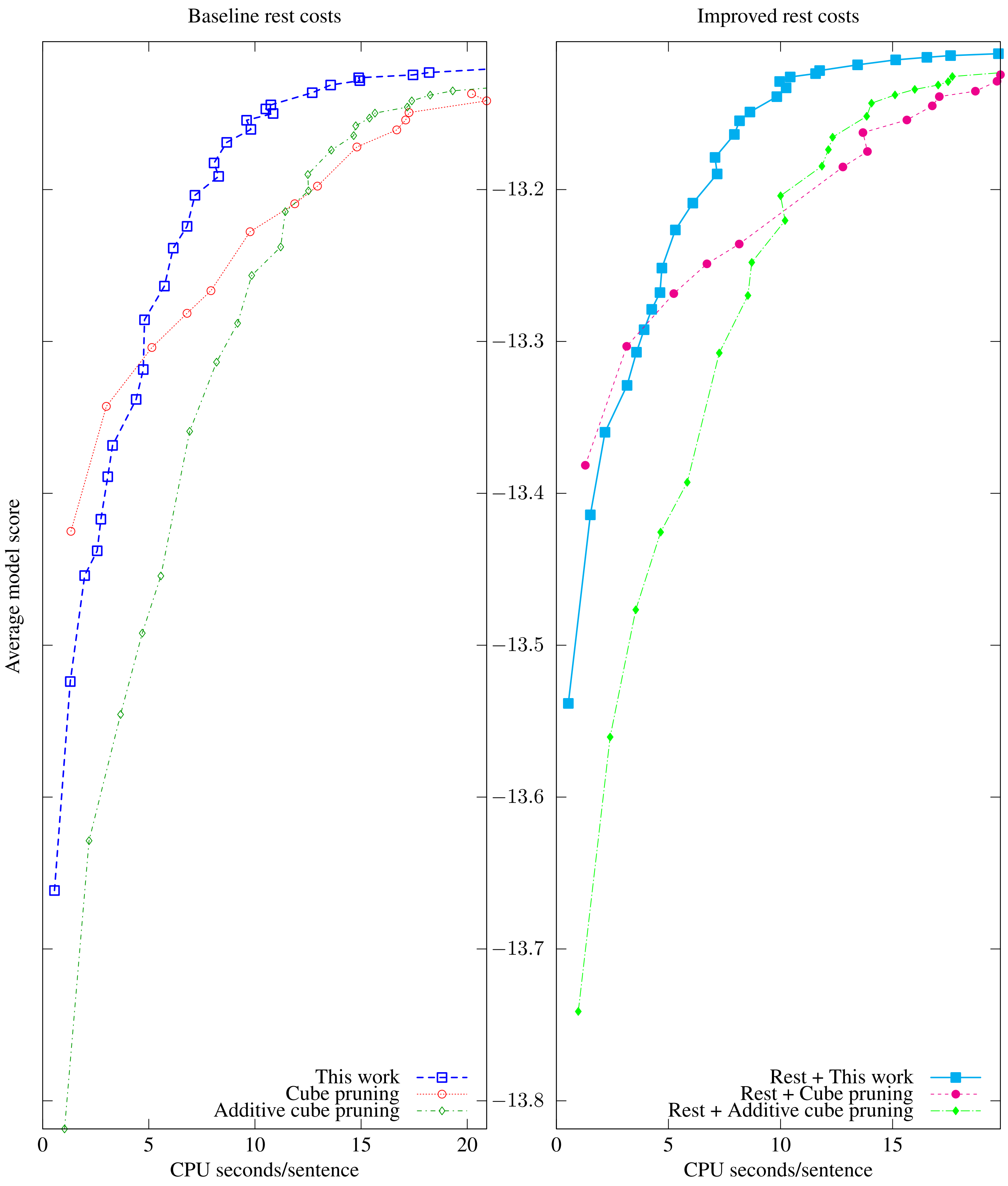


Figure A.5: Time-accuracy trade-off for **Moses on English–German with target syntax**.

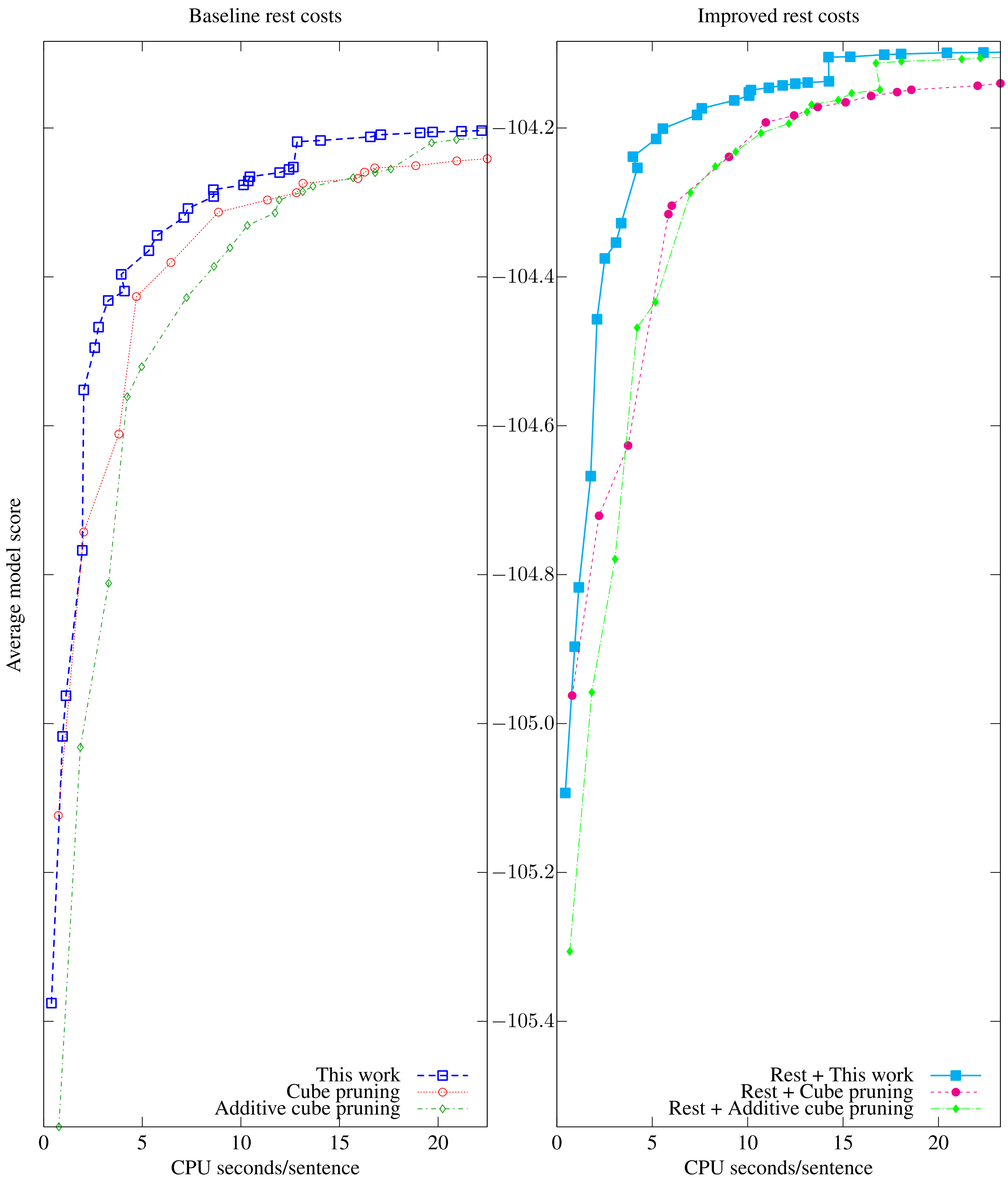


Figure A.6: Time-accuracy trade-off for **Moses on German–English with target syntax**. As noted at the end of Section 7.4.3, the model score jumps when a way is found to translate “durchzogenen” as “criss-crossed”.

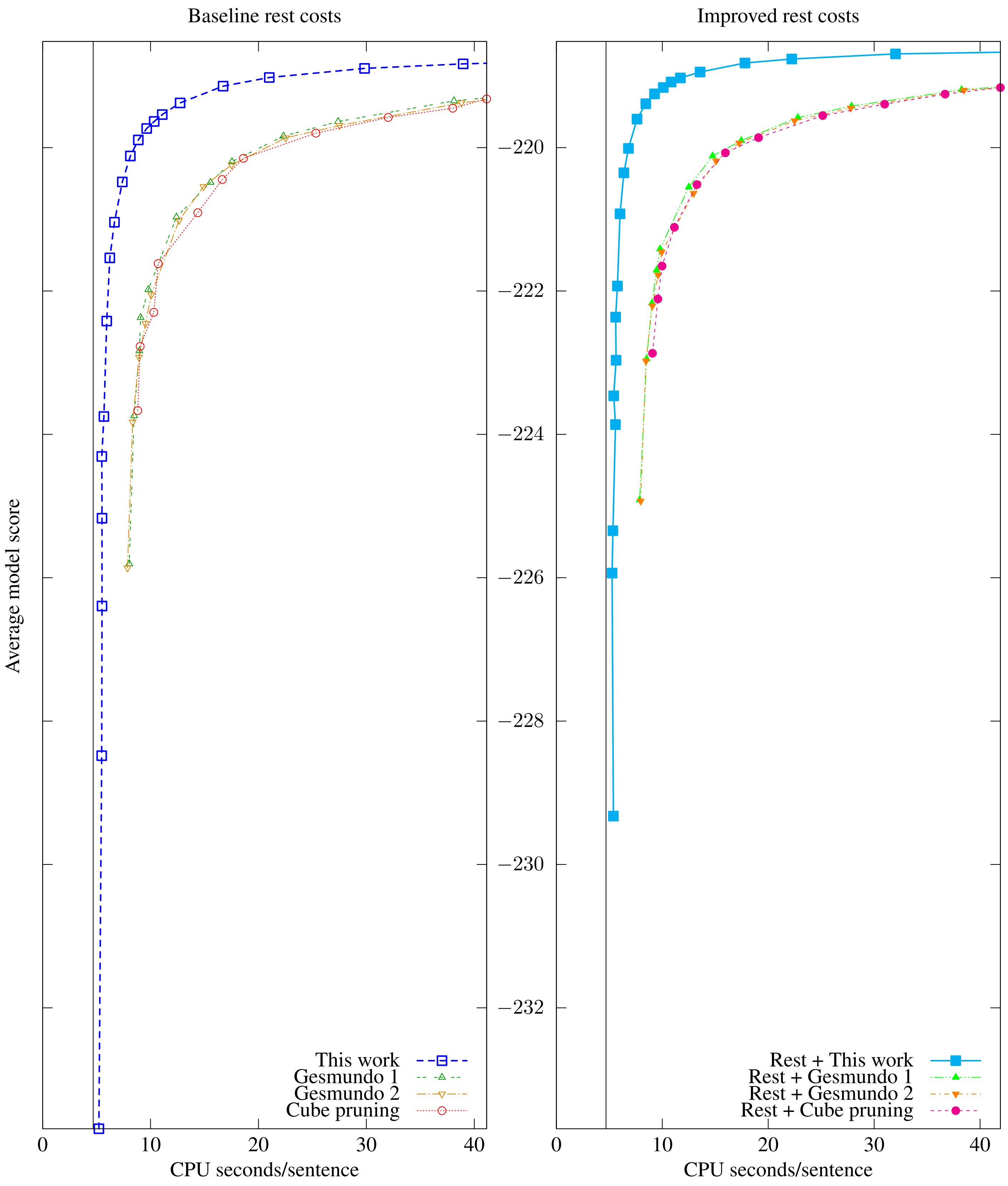


Figure A.7: Time-accuracy trade-off for **cdec on tree-to-tree French–English**. Due to computational costs, cube pruning was run with beam sizes up to 300. The black line indicates parsing cost.

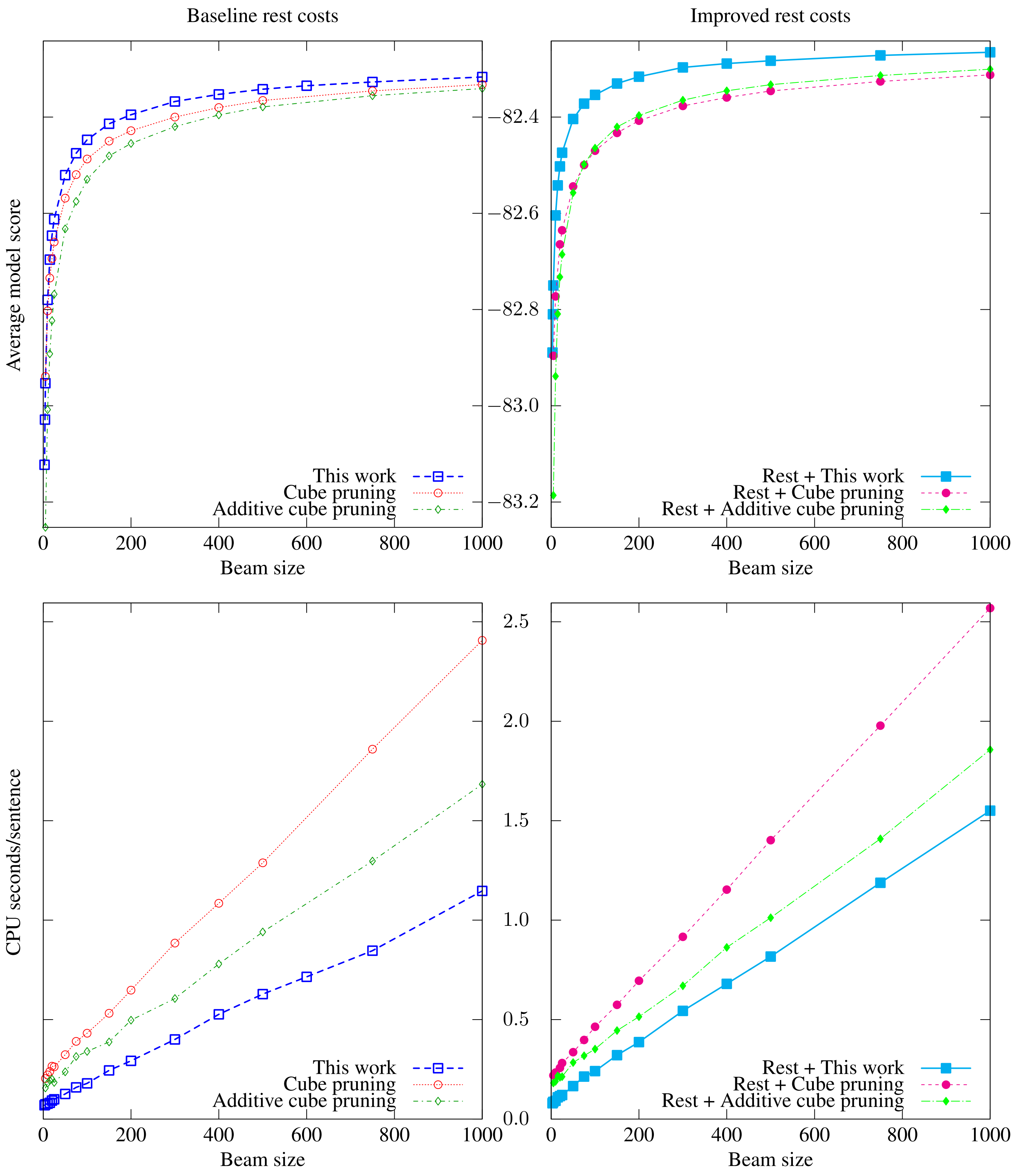


Figure A.8: Performance of **Moses on hierarchical Chinese–English** translation by beam size.

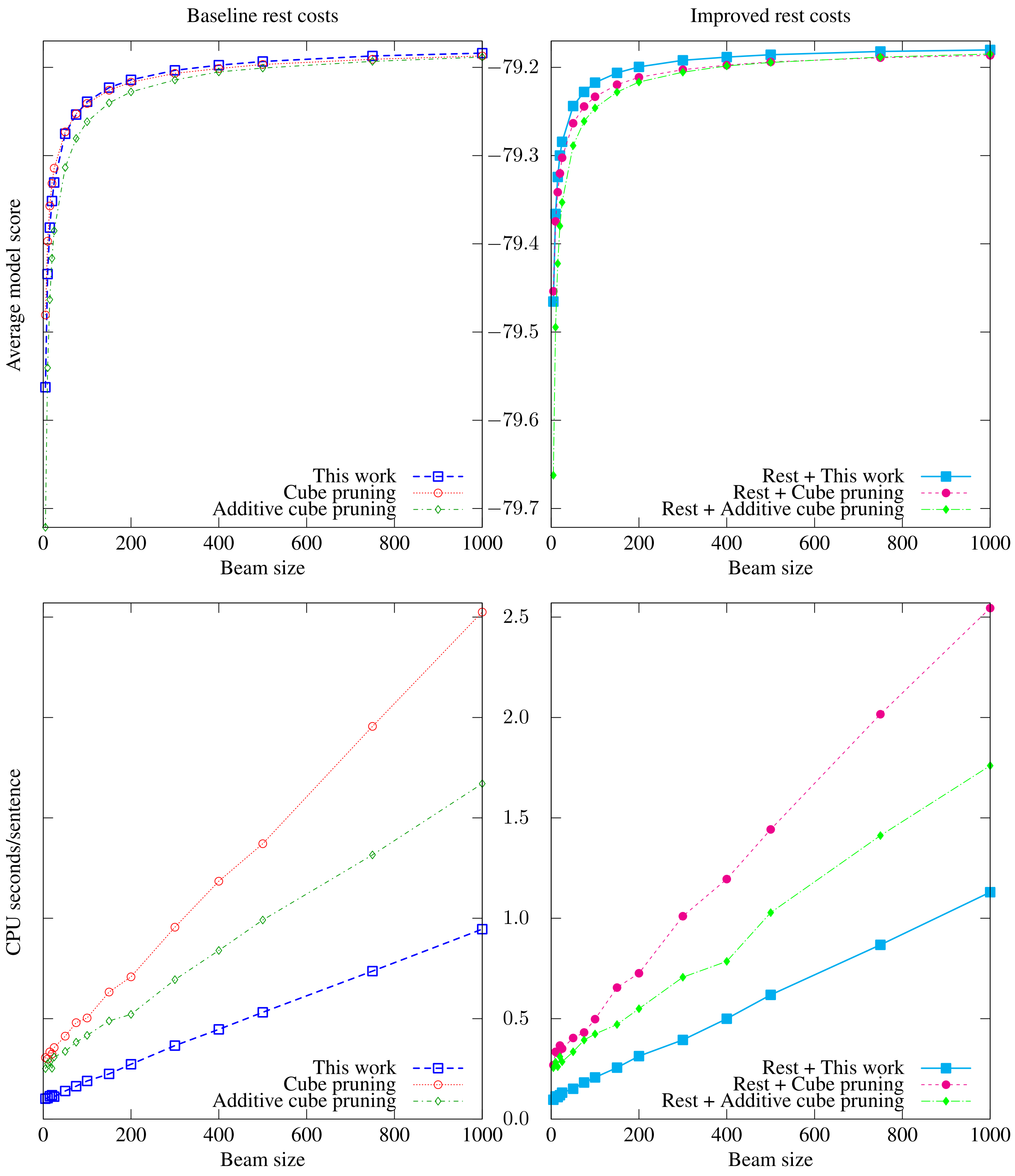


Figure A.9: Performance of **Moses on hierarchical English–German** translation by beam size.

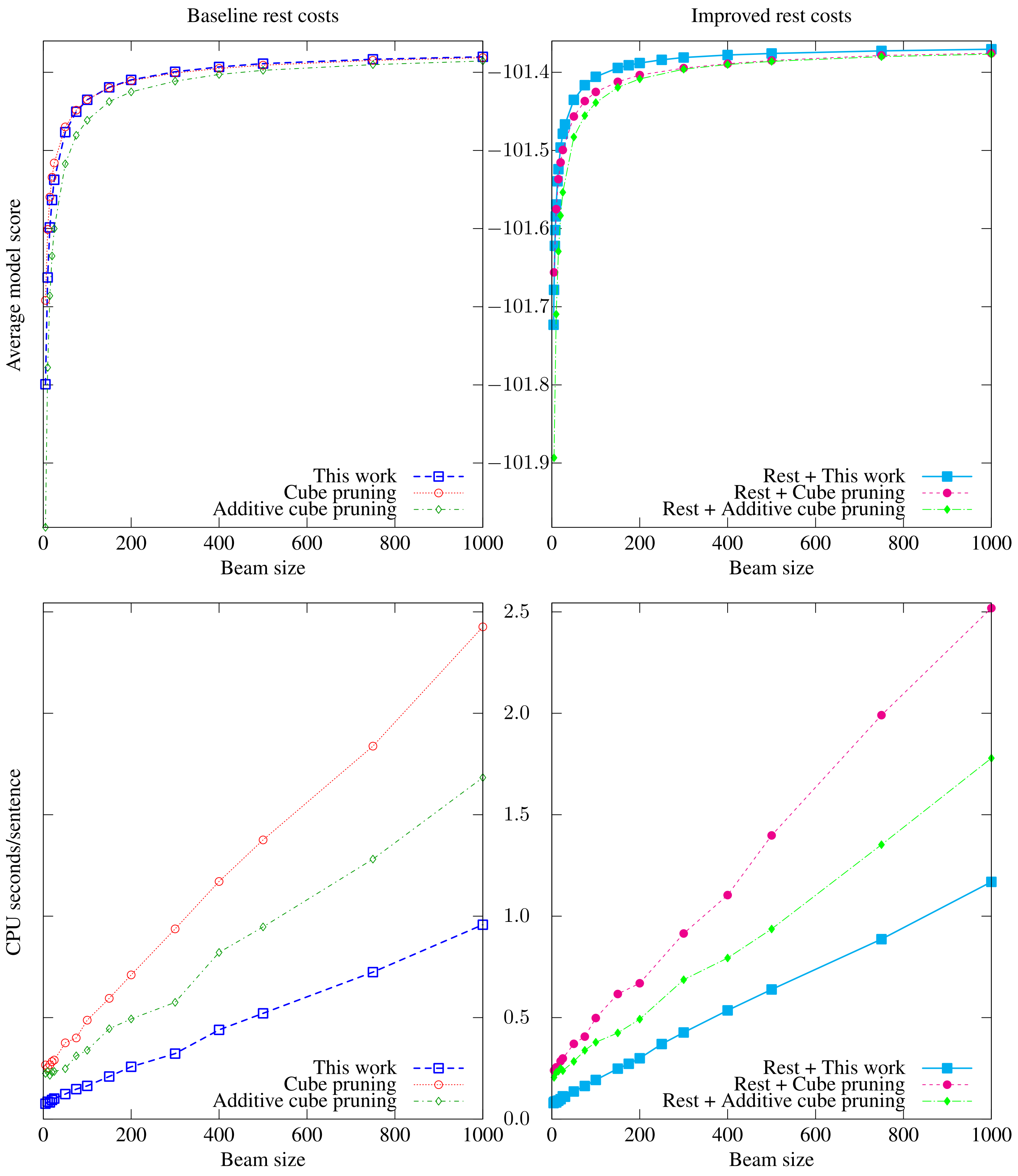


Figure A.10: Performance of **Moses on hierarchical German–English** translation by beam size.

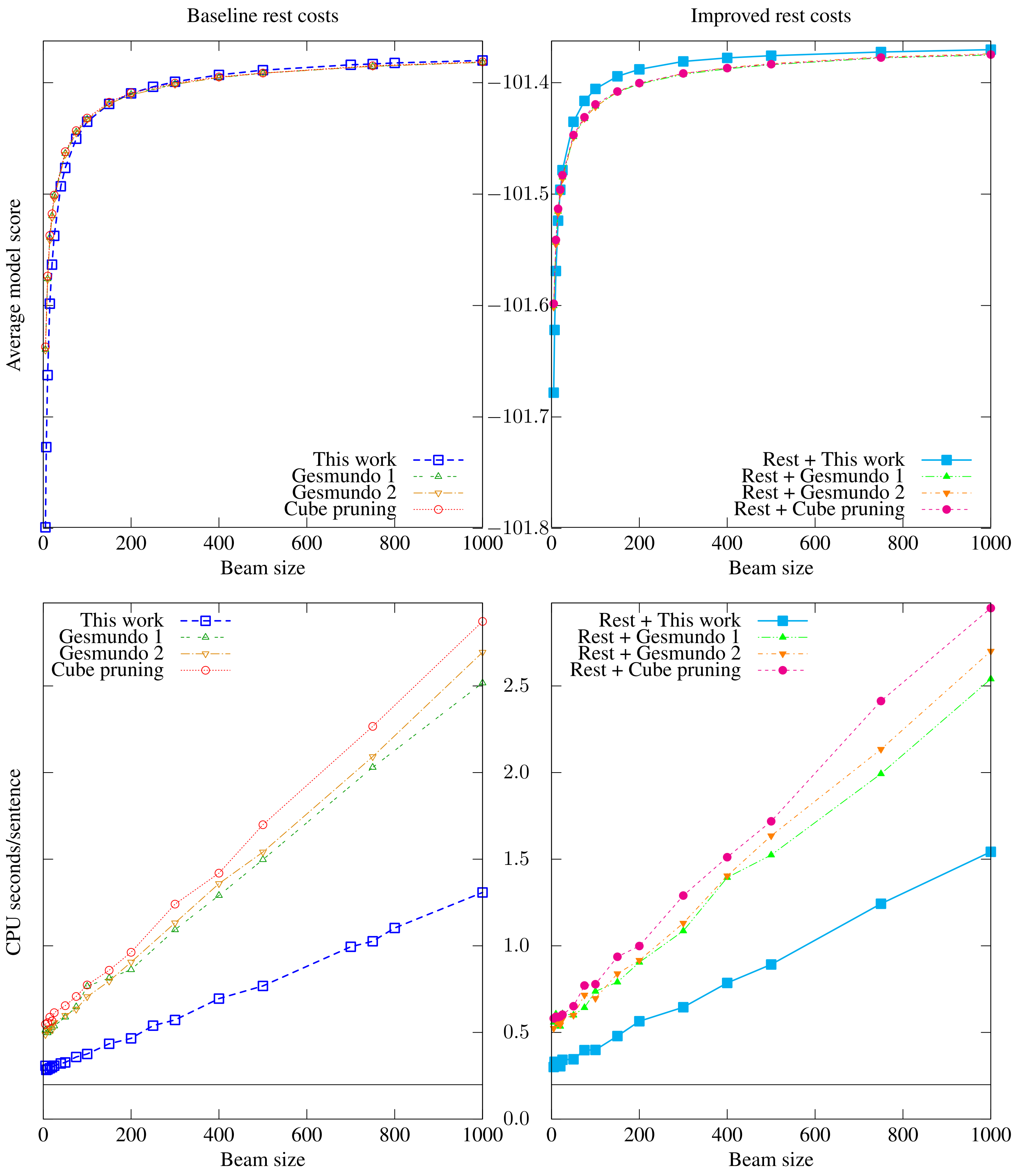


Figure A.11: Performance of **cdec on hierarchical German–English** translation by beam size. The black line indicates parsing cost.

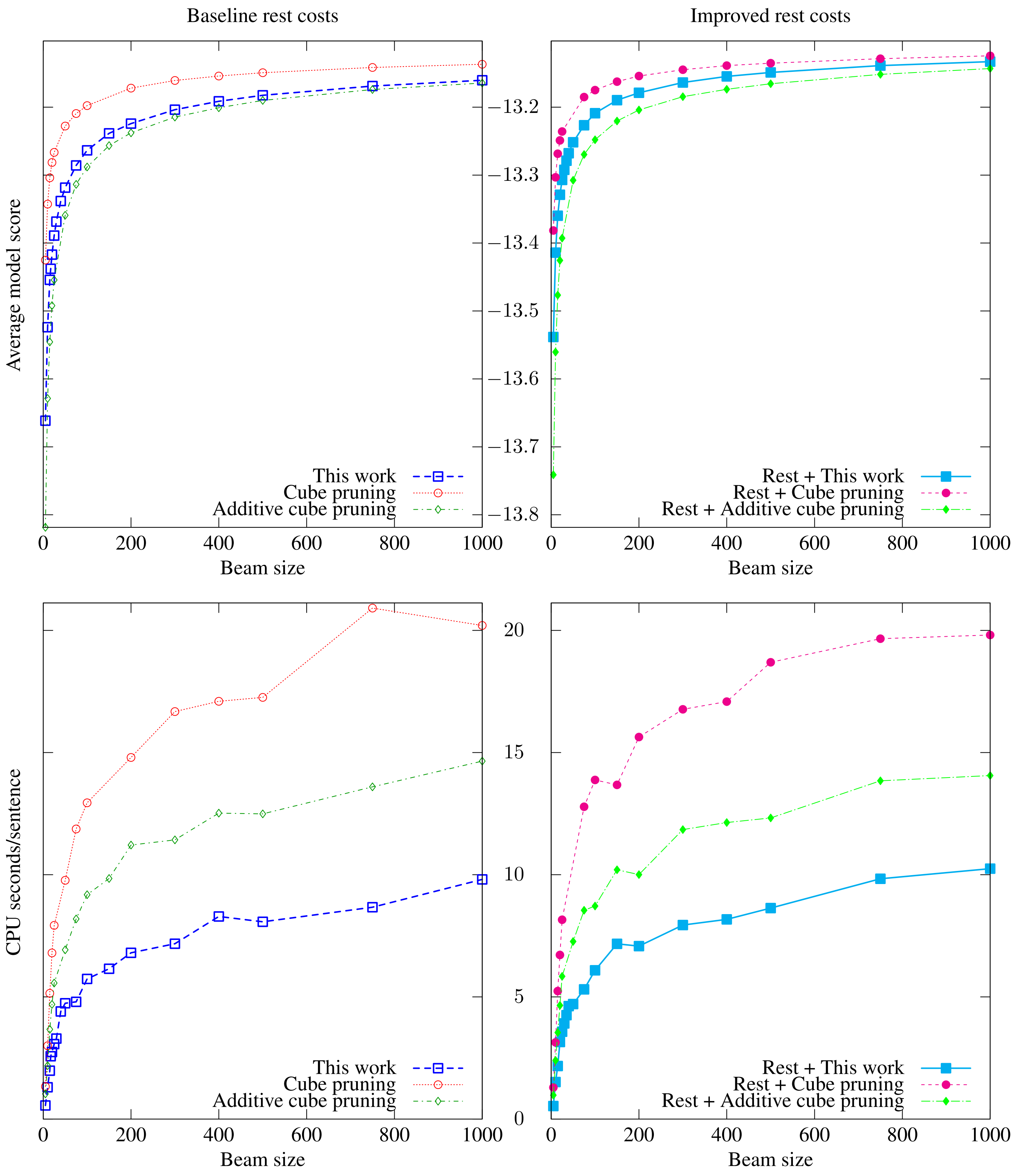


Figure A.12: Performance of **Moses on English–German with target syntax** translation by beam size.

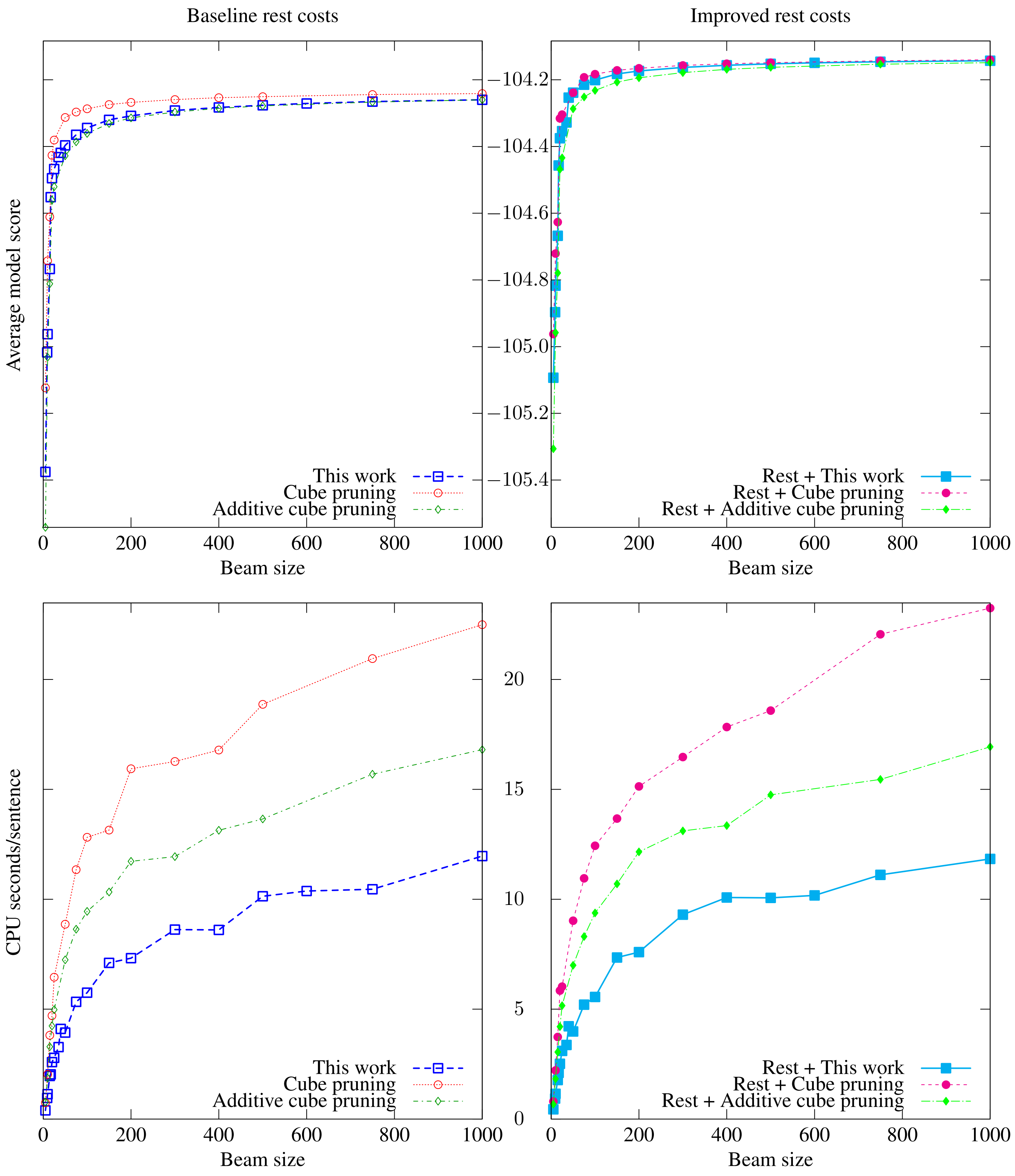


Figure A.13: Performance of **Moses on German–English with target syntax** translation by beam size.

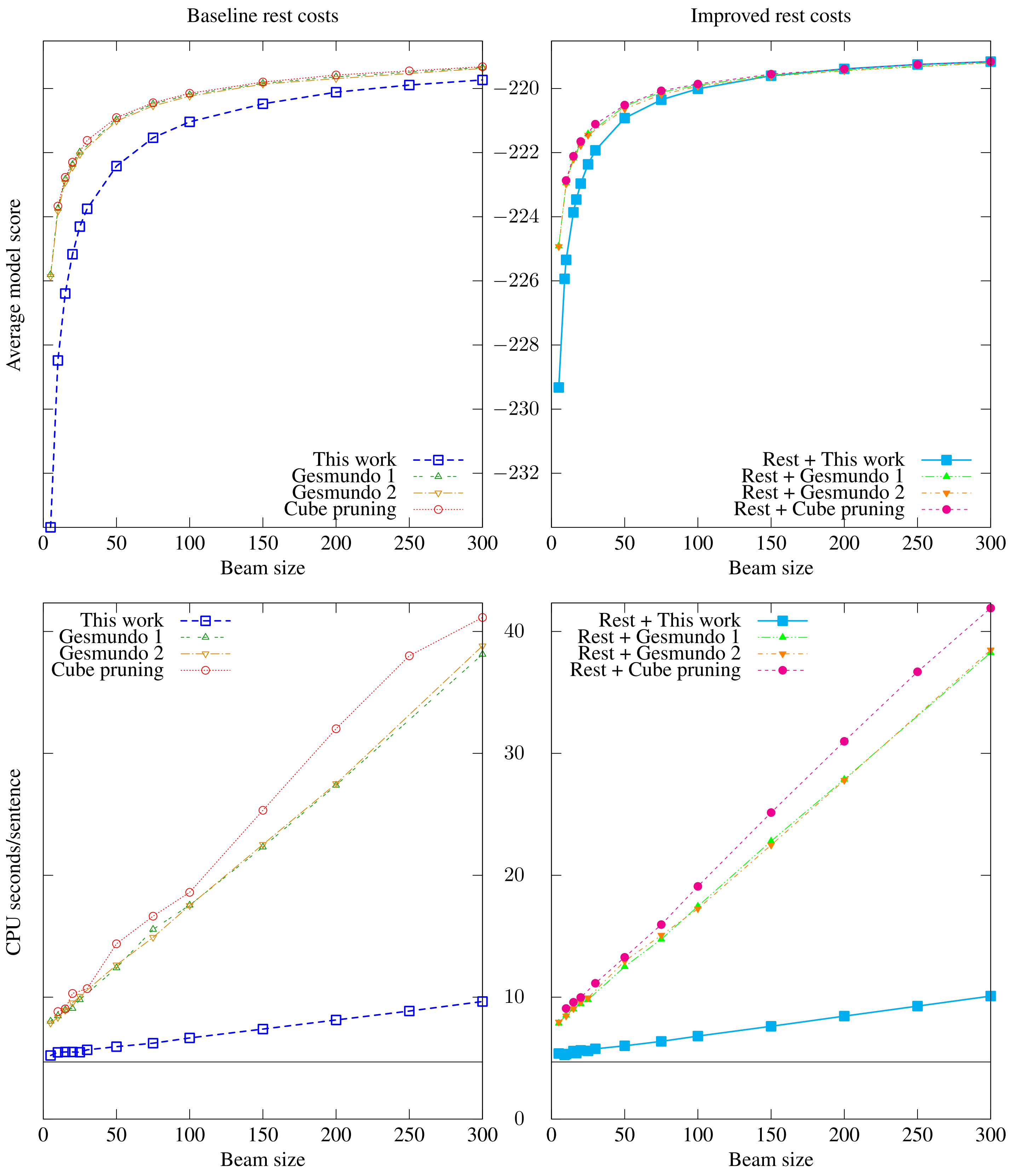


Figure A.14: Performance of **cdec on tree-to-tree French–English** translation by beam size. Due to computational costs, cube pruning was run with beam sizes up to 300. The black line indicates parsing cost.

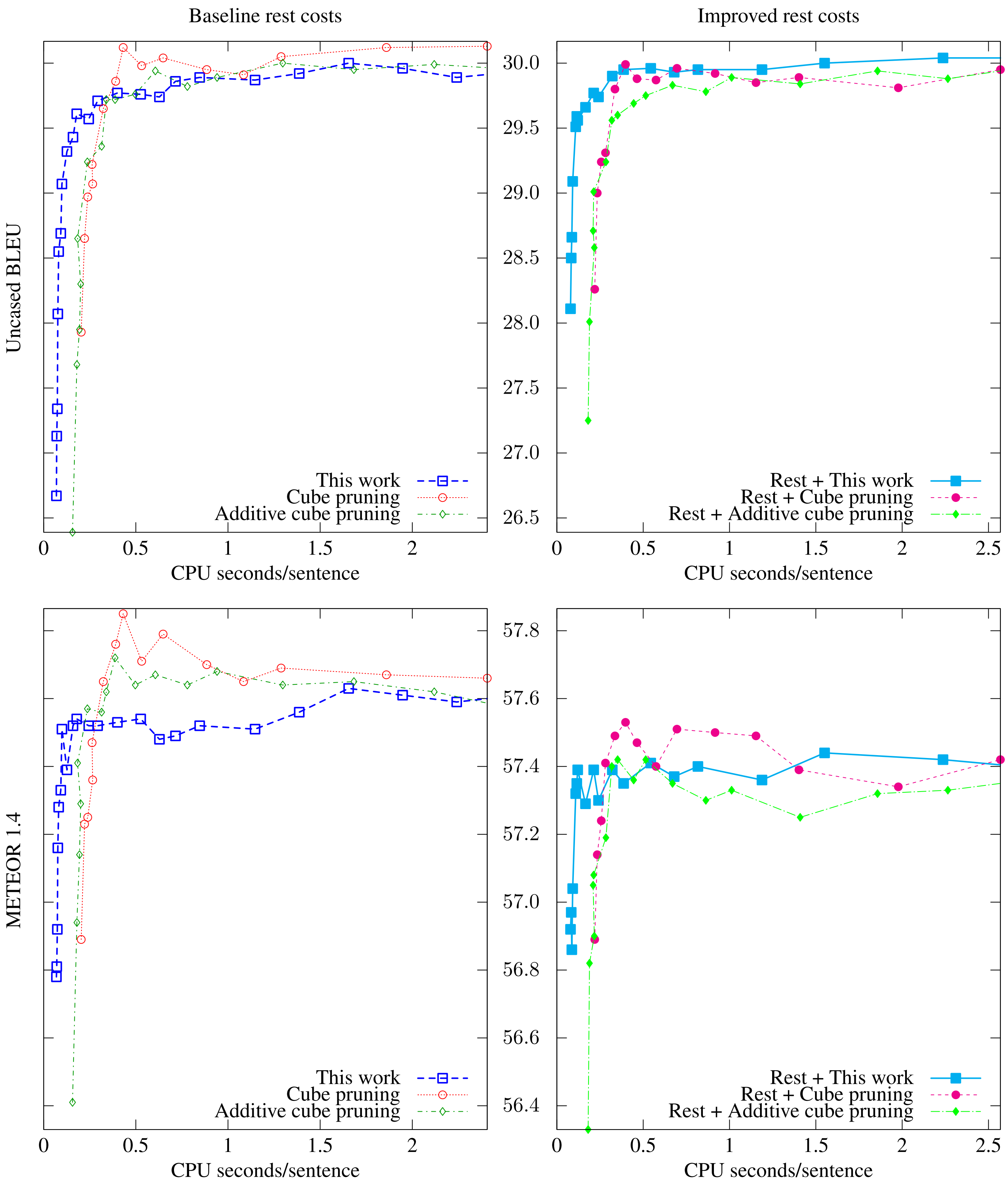


Figure A.15: Time-quality trade-offs for **Moses on hierarchical Chinese–English**. The test set has four references.

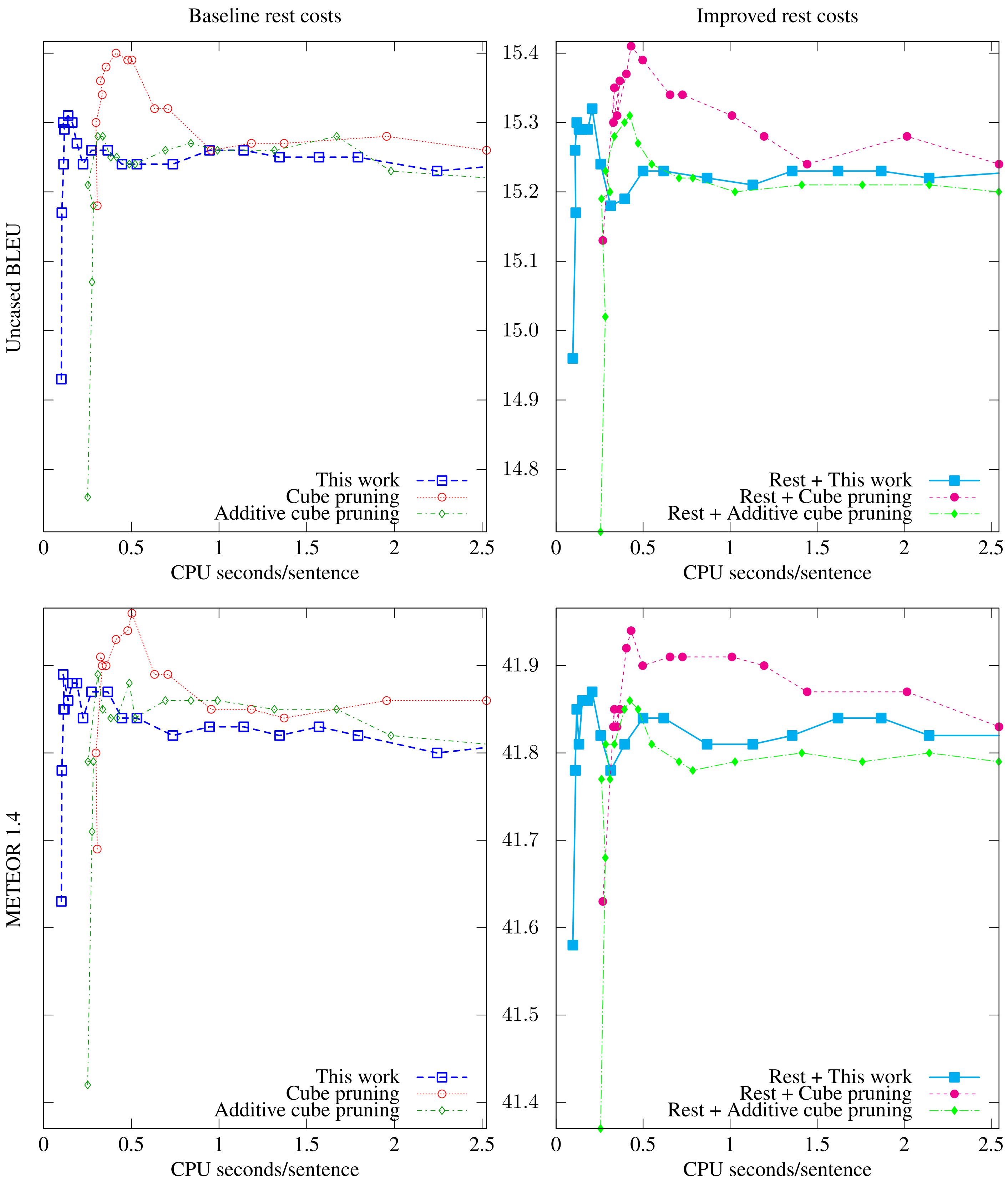


Figure A.16: Time-quality trade-offs for **Moses on hierarchical English–German**.

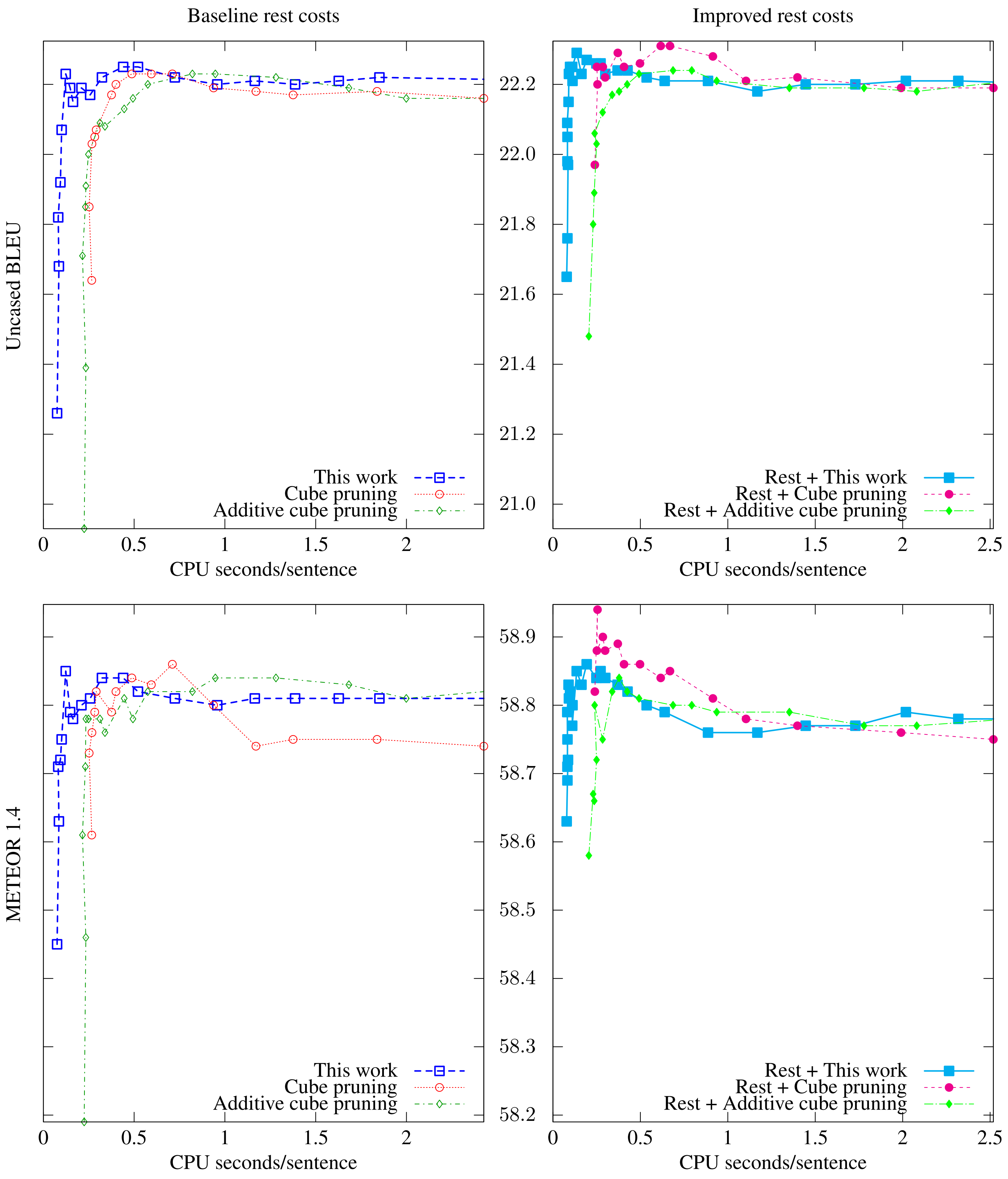


Figure A.17: Time-quality trade-offs for **Moses on hierarchical German–English**.

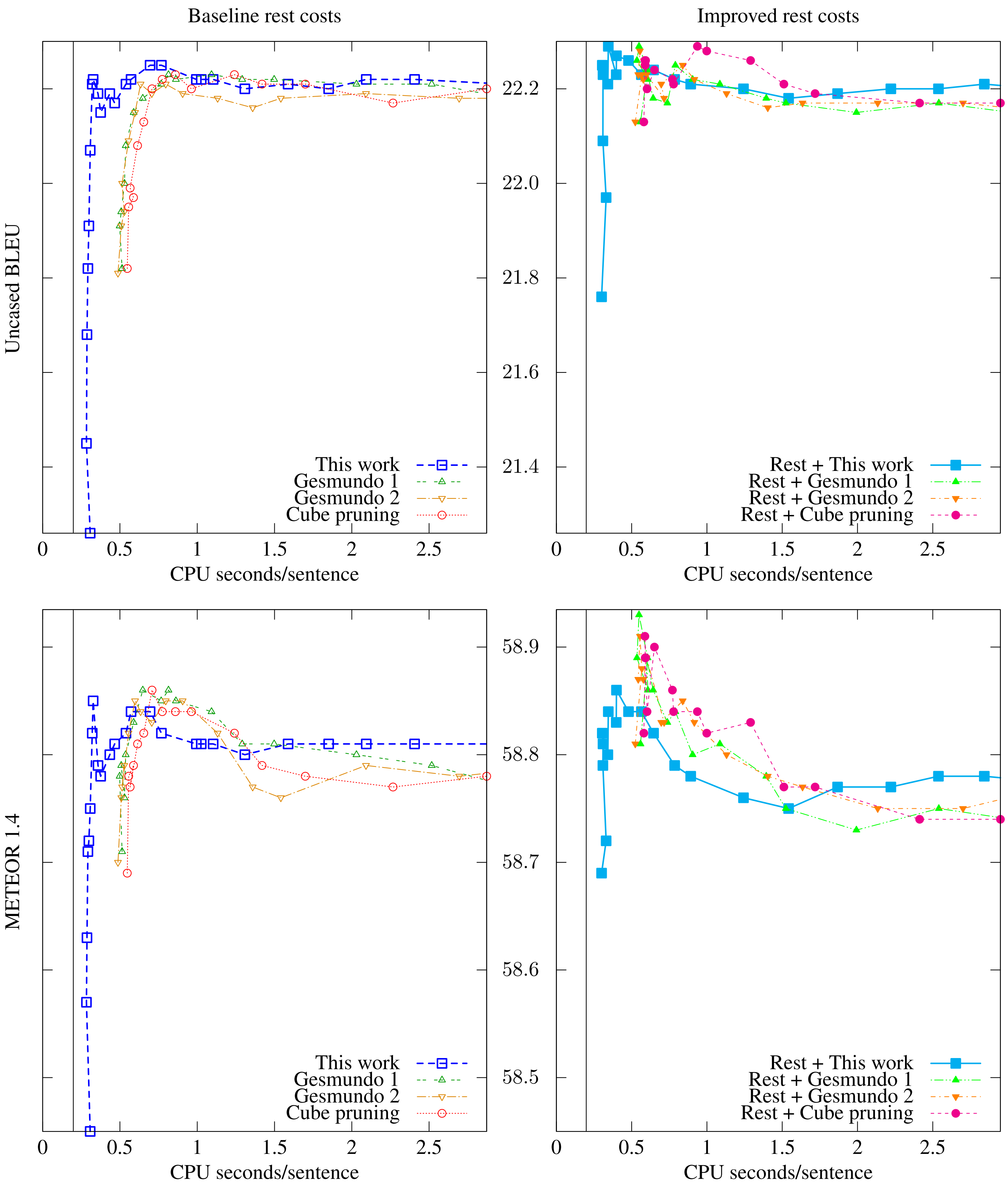


Figure A.18: Time-quality trade-offs for **cdec on hierarchical German–English**. The black line indicates parsing cost.

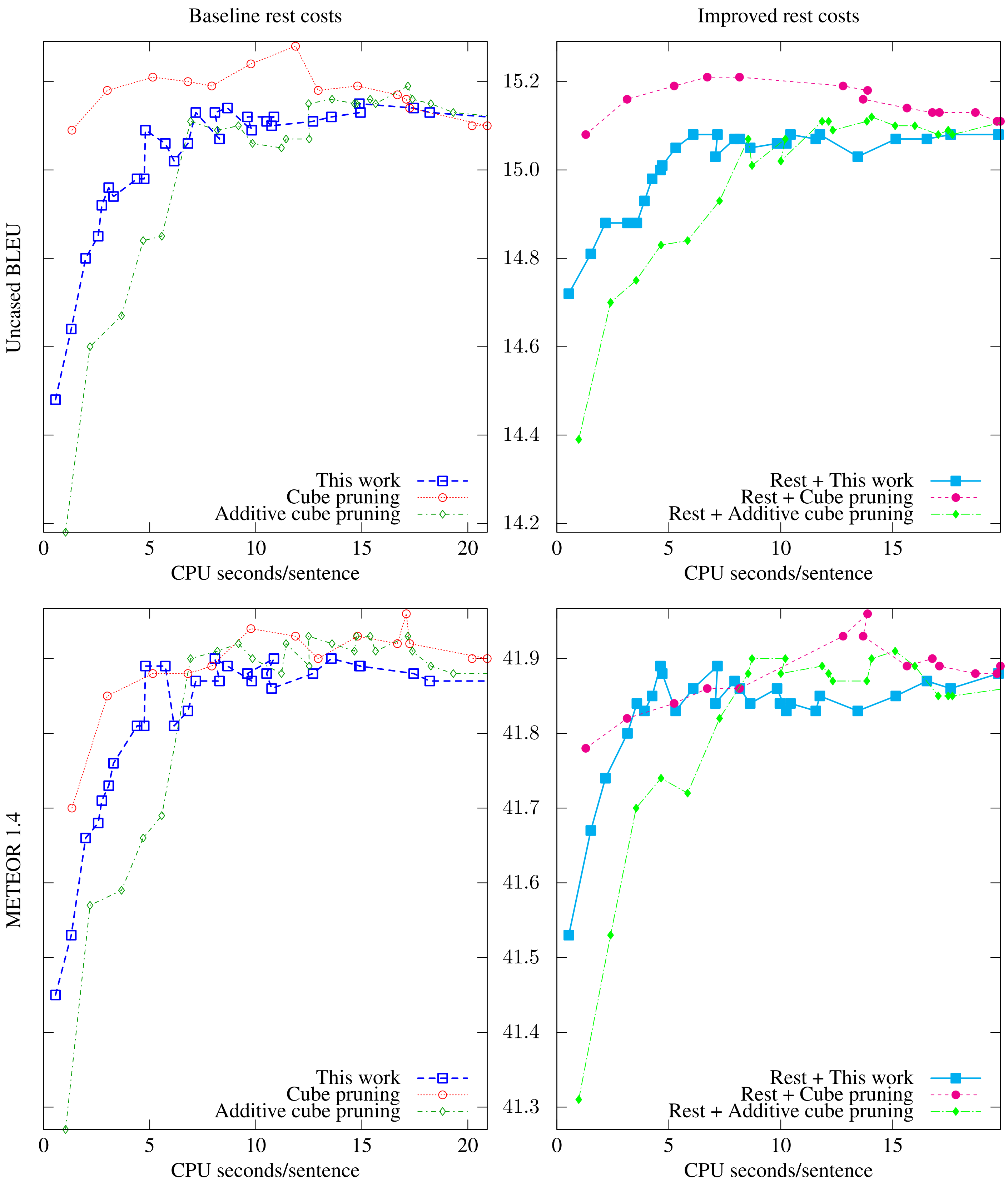


Figure A.19: Time-quality trade-offs for **Moses on English–German with target syntax**.

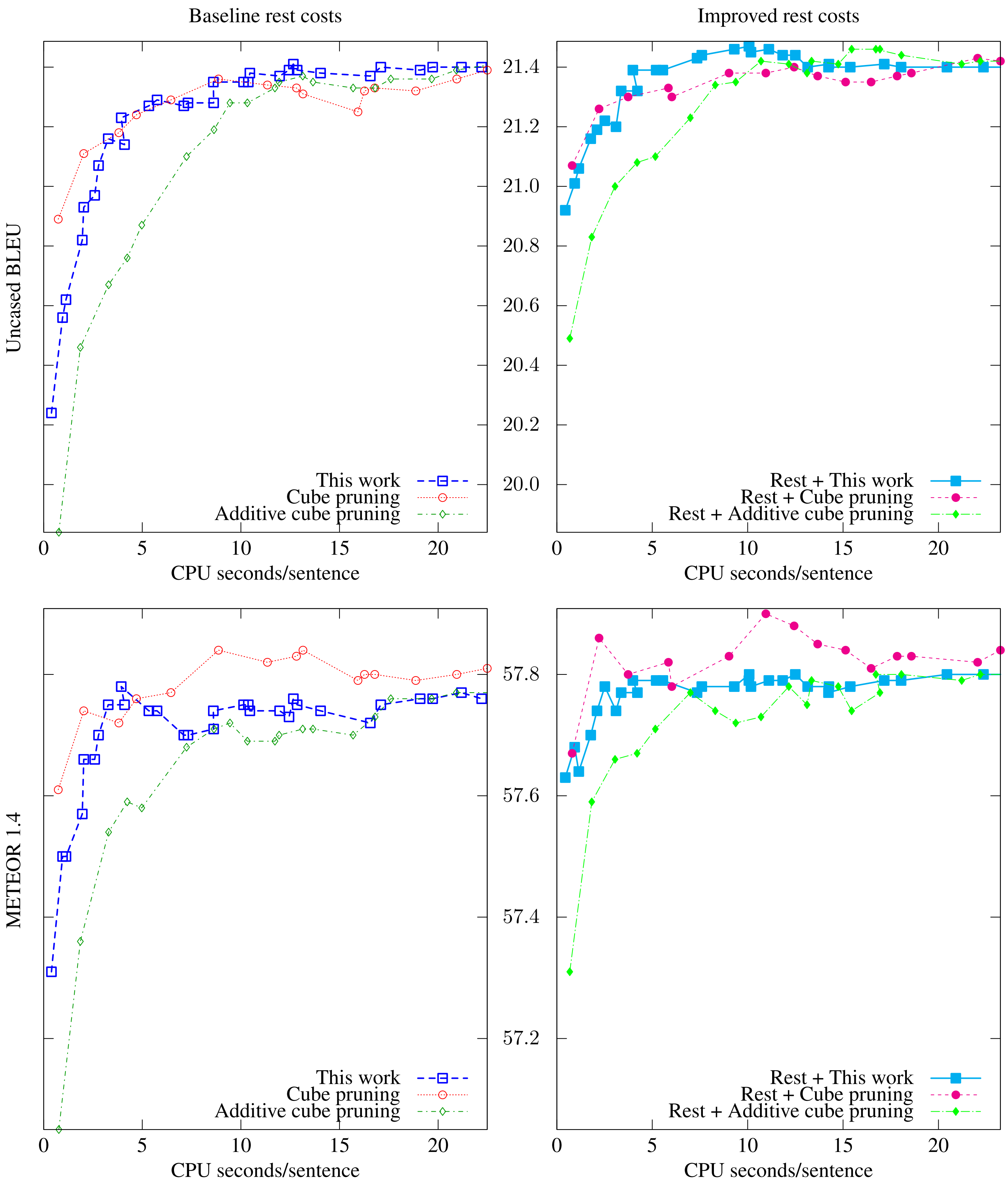


Figure A.20: Time-quality trade-offs for **Moses on German–English with target syntax**.

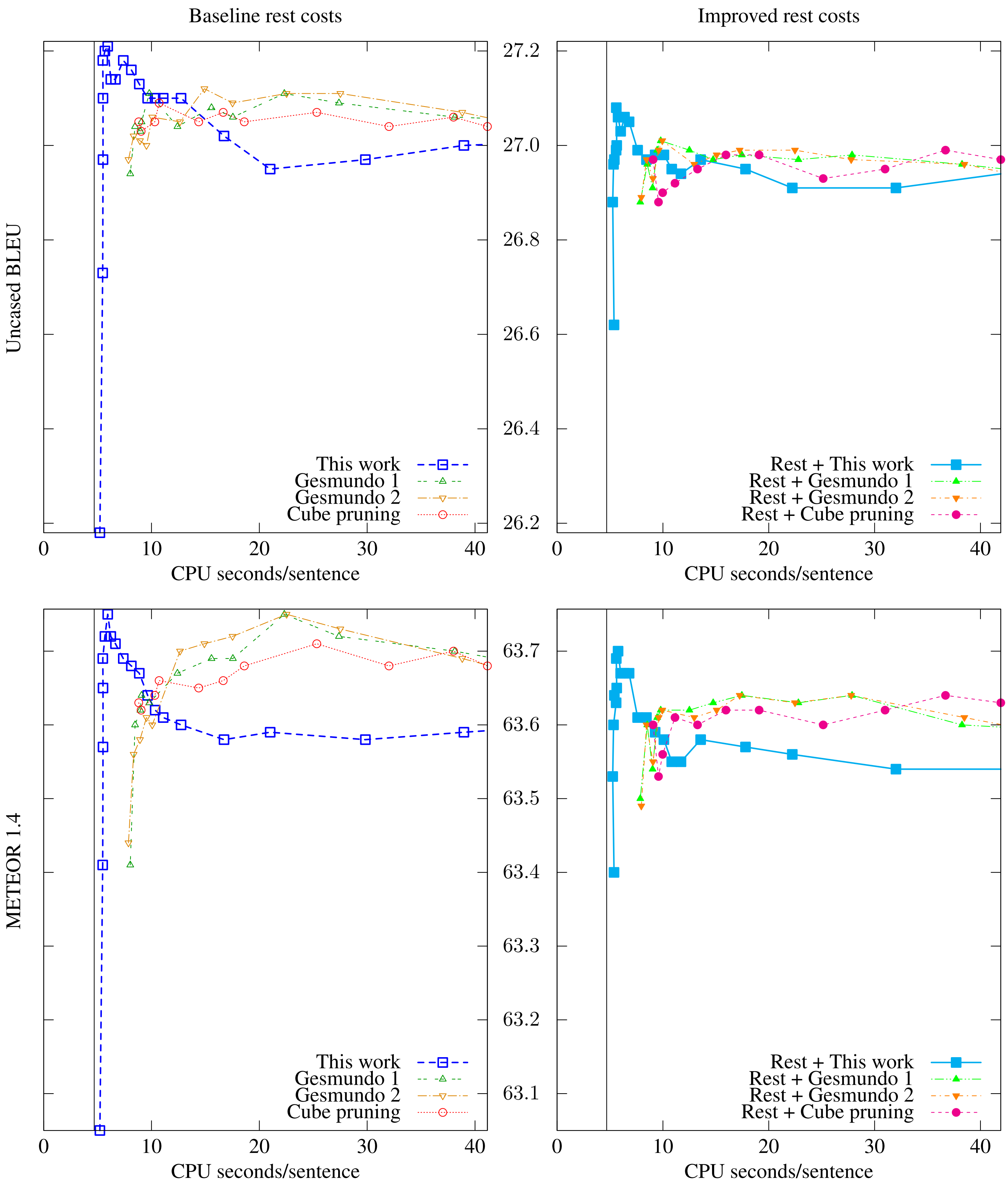


Figure A.21: Time-quality trade-offs for **cdec on tree-to-tree French–English**. Due to computational costs, cube pruning was run with beam sizes up to 300. The black line indicates parsing cost.

	This work			**Cube pruning**			**Additive cube**		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.073	−82.953	27.34	0.204	−82.939	27.93	0.156	−83.253	26.39
10	0.076	−82.780	28.07	0.222	−82.803	28.65	0.181	−83.008	27.68
15	0.114	−82.696	28.55	0.239	−82.735	28.97	0.195	−82.892	27.95
20	0.092	−82.646	28.69	0.266	−82.694	29.07	0.205	−82.823	28.30
25	0.098	−82.612	29.07	0.262	−82.660	29.22	0.184	−82.768	28.65
50	0.125	−82.521	29.32	0.324	−82.569	29.65	0.237	−82.632	29.24
75	0.158	−82.475	29.43	0.391	−82.520	29.86	0.314	−82.575	29.36
100	0.179	−82.447	29.61	0.431	−82.487	30.12	0.341	−82.530	29.72
150	0.244	−82.414	29.57	0.532	−82.450	29.98	0.387	−82.481	29.72
200	0.292	−82.395	29.71	0.648	−82.428	30.04	0.498	−82.455	29.76
300	0.400	−82.368	29.77	0.885	−82.400	29.95	0.657	−82.420	29.94
400	0.526	−82.353	29.76	1.085	−82.380	29.91	0.780	−82.396	29.82
500	0.627	−82.342	29.74	1.288	−82.366	30.05	0.941	−82.379	29.89
750	0.847	−82.327	29.89	1.860	−82.346	30.12	1.298	−82.356	30.00
1000	1.146	−82.317	29.87	2.407	−82.332	30.13	1.684	−82.341	29.95

	Rest+This work			**Rest+Cube pruning**			**Rest+Additive cube**		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.087	−82.750	28.66	0.218	−82.896	28.26	0.182	−83.186	27.25
10	0.092	−82.604	29.09	0.234	−82.773	29.00	0.190	−82.939	28.01
20	0.116	−82.503	29.59	0.257	−82.665	29.24	0.211	−82.732	28.71
25	0.122	−82.474	29.56	0.281	−82.635	29.31	0.214	−82.686	29.01
50	0.224	−82.404	29.66	0.336	−82.544	29.80	0.284	−82.557	29.24
75	0.214	−82.372	29.77	0.397	−82.500	29.99	0.320	−82.498	29.56
100	0.242	−82.354	29.74	0.463	−82.470	29.88	0.352	−82.465	29.60
150	0.322	−82.330	29.90	0.574	−82.433	29.87	0.447	−82.421	29.69
200	0.387	−82.316	29.95	0.695	−82.408	29.96	0.515	−82.397	29.75
300	0.545	−82.297	29.96	0.916	−82.377	29.92	0.670	−82.365	29.83
400	0.680	−82.289	29.93	1.154	−82.359	29.85	0.864	−82.345	29.78
500	0.818	−82.283	29.95	1.401	−82.346	29.89	1.013	−82.333	29.89
750	1.189	−82.272	29.95	1.978	−82.326	29.81	1.409	−82.314	29.84
1000	1.552	−82.265	30.00	2.569	−82.312	29.95	1.858	−82.300	29.94

Table A.1: Numerical results for **Moses on hierarchical Chinese–English** at select beam sizes. The test set has four references.

	This work			Cube pruning			Additive cube		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.101	−79.563	14.93	0.306	−79.481	15.18	0.251	−79.721	14.76
10	0.104	−79.434	15.17	0.299	−79.397	15.30	0.276	−79.541	15.07
15	0.113	−79.382	15.24	0.334	−79.357	15.34	0.284	−79.463	15.18
20	0.119	−79.352	15.29	0.324	−79.332	15.36	0.253	−79.416	15.21
25	0.112	−79.331	15.30	0.356	−79.314	15.38	0.309	−79.385	15.28
50	0.139	−79.275	15.31	0.413	−79.273	15.40	0.337	−79.314	15.28
75	0.163	−79.253	15.30	0.480	−79.253	15.39	0.383	−79.281	15.25
100	0.189	−79.239	15.27	0.504	−79.241	15.39	0.417	−79.262	15.25
150	0.224	−79.223	15.24	0.633	−79.226	15.32	0.489	−79.240	15.24
200	0.272	−79.214	15.26	0.708	−79.217	15.32	0.521	−79.228	15.24
300	0.366	−79.203	15.26	0.956	−79.207	15.26	0.694	−79.214	15.26
400	0.446	−79.198	15.24	1.184	−79.201	15.27	0.840	−79.205	15.27
500	0.531	−79.193	15.24	1.372	−79.197	15.27	0.991	−79.201	15.26
750	0.737	−79.187	15.24	1.956	−79.191	15.28	1.316	−79.193	15.26
1000	0.945	−79.184	15.26	2.525	−79.187	15.26	1.671	−79.188	15.28

	Rest+This work			Rest+Cube pruning			Rest+Additive cube		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.097	−79.465	14.96	0.269	−79.454	15.13	0.256	−79.662	14.71
10	0.113	−79.366	15.17	0.335	−79.375	15.35	0.283	−79.495	15.02
15	0.109	−79.324	15.26	0.330	−79.341	15.30	0.261	−79.422	15.19
20	0.119	−79.300	15.30	0.366	−79.320	15.36	0.311	−79.380	15.20
25	0.132	−79.284	15.29	0.350	−79.302	15.31	0.285	−79.353	15.23
50	0.151	−79.244	15.29	0.404	−79.263	15.37	0.335	−79.289	15.28
75	0.182	−79.228	15.29	0.431	−79.244	15.41	0.393	−79.261	15.30
100	0.208	−79.217	15.32	0.497	−79.233	15.39	0.424	−79.246	15.31
150	0.257	−79.206	15.24	0.655	−79.220	15.34	0.471	−79.228	15.27
200	0.314	−79.199	15.18	0.726	−79.211	15.34	0.550	−79.217	15.24
300	0.394	−79.192	15.19	1.011	−79.203	15.31	0.707	−79.205	15.22
400	0.500	−79.188	15.23	1.195	−79.198	15.28	0.785	−79.198	15.22
500	0.623	−79.186	15.23	1.442	−79.194	15.24	1.028	−79.194	15.20
750	0.868	−79.182	15.22	2.017	−79.189	15.28	1.412	−79.188	15.21
1000	1.131	−79.180	15.21	2.546	−79.186	15.24	1.760	−79.185	15.21

Table A.2: Numerical results for **Moses on hierarchical English–German** at select beam sizes.

	This work			Cube pruning			Additive cube		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.075	−101.799	21.26	0.267	−101.692	21.64	0.224	−101.982	20.93
10	0.085	−101.663	21.68	0.252	−101.601	21.85	0.234	−101.778	21.39
15	0.082	−101.598	21.82	0.268	−101.560	22.03	0.215	−101.686	21.71
20	0.094	−101.563	21.92	0.283	−101.534	22.05	0.231	−101.635	21.85
25	0.104	−101.537	22.07	0.291	−101.516	22.07	0.234	−101.600	21.91
50	0.123	−101.477	22.23	0.375	−101.470	22.17	0.249	−101.517	22.00
75	0.147	−101.450	22.19	0.400	−101.448	22.20	0.311	−101.481	22.09
100	0.163	−101.435	22.15	0.487	−101.435	22.23	0.339	−101.461	22.08
150	0.209	−101.419	22.19	0.595	−101.420	22.23	0.445	−101.437	22.13
200	0.257	−101.410	22.17	0.711	−101.411	22.23	0.493	−101.425	22.16
300	0.323	−101.399	22.22	0.938	−101.401	22.19	0.575	−101.411	22.20
400	0.439	−101.393	22.25	1.171	−101.395	22.18	0.821	−101.403	22.23
500	0.521	−101.389	22.25	1.376	−101.391	22.17	0.947	−101.397	22.23
750	0.724	−101.383	22.22	1.838	−101.385	22.18	1.281	−101.390	22.22
1000	0.958	−101.380	22.20	2.426	−101.381	22.16	1.683	−101.386	22.19
1250	1.164	−101.378	22.21	2.922	−101.379	22.17	1.999	−101.382	22.16
1500	1.387	−101.376	22.20	3.554	−101.378	22.19	2.429	−101.380	22.16

	Rest+This work			Rest+Cube pruning			Rest+Additive cube		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.083	−101.678	21.76	0.239	−101.656	21.97	0.205	−101.893	21.48
10	0.082	−101.569	22.09	0.255	−101.575	22.20	0.229	−101.710	21.80
15	0.091	−101.524	22.23	0.251	−101.537	22.25	0.236	−101.629	21.89
20	0.098	−101.496	22.25	0.285	−101.515	22.25	0.249	−101.583	22.03
25	0.112	−101.479	22.21	0.298	−101.499	22.22	0.238	−101.554	22.06
50	0.136	−101.435	22.29	0.370	−101.457	22.29	0.283	−101.483	22.12
75	0.163	−101.416	22.23	0.406	−101.437	22.25	0.338	−101.455	22.17
100	0.193	−101.406	22.27	0.498	−101.425	22.26	0.378	−101.439	22.18
150	0.249	−101.394	22.26	0.616	−101.412	22.31	0.425	−101.419	22.20
200	0.300	−101.388	22.23	0.669	−101.404	22.31	0.492	−101.408	22.23
300	0.427	−101.381	22.24	0.914	−101.395	22.28	0.687	−101.396	22.24
400	0.536	−101.378	22.22	1.133	−101.389	22.21	0.793	−101.390	22.24
500	0.639	−101.376	22.21	1.398	−101.385	22.22	0.936	−101.386	22.21
750	0.887	−101.373	22.21	1.991	−101.378	22.19	1.352	−101.380	22.19
1000	1.169	−101.370	22.18	2.519	−101.376	22.19	1.779	−101.377	22.19
1250	1.446	−101.370	22.20	3.168	−101.374	22.18	2.081	−101.375	22.18
1500	1.730	−101.369	22.20	3.727	−101.373	22.18	2.620	−101.373	22.21

Table A.3: Numerical results for **Moses on hierarchical German–English** at select beam sizes.

	This work			**Cube pruning**			**Gesmundo 1**			**Gesmundo 2**		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.307	−101.799	21.26	0.547	−101.637	21.82	0.515	−101.639	21.82	0.488	−101.640	21.81
10	0.286	−101.662	21.68	0.556	−101.574	21.95	0.498	−101.576	21.91	0.509	−101.577	21.91
15	0.292	−101.598	21.82	0.588	−101.537	21.97	0.508	−101.539	21.94	0.528	−101.541	21.94
20	0.299	−101.563	21.91	0.566	−101.518	21.99	0.530	−101.519	22.00	0.514	−101.521	22.00
25	0.308	−101.537	22.07	0.614	−101.501	22.08	0.537	−101.502	22.08	0.556	−101.504	22.09
50	0.326	−101.476	22.22	0.654	−101.462	22.13	0.589	−101.463	22.15	0.598	−101.464	22.15
75	0.358	−101.450	22.19	0.707	−101.443	22.20	0.648	−101.444	22.18	0.633	−101.445	22.21
100	0.375	−101.435	22.15	0.773	−101.432	22.22	0.767	−101.433	22.21	0.707	−101.433	22.19
150	0.434	−101.419	22.19	0.859	−101.418	22.23	0.814	−101.419	22.23	0.796	−101.419	22.21
200	0.465	−101.410	22.17	0.962	−101.409	22.20	0.862	−101.411	22.22	0.908	−101.411	22.19
300	0.573	−101.399	22.22	1.241	−101.400	22.23	1.094	−101.401	22.23	1.132	−101.402	22.18
400	0.696	−101.393	22.25	1.419	−101.395	22.21	1.291	−101.396	22.22	1.359	−101.395	22.16
500	0.768	−101.389	22.25	1.698	−101.391	22.21	1.498	−101.391	22.22	1.541	−101.391	22.18
750	1.026	−101.383	22.22	2.266	−101.385	22.17	2.030	−101.385	22.21	2.092	−101.385	22.19
1000	1.308	−101.380	22.20	2.873	−101.381	22.20	2.518	−101.382	22.21	2.695	−101.382	22.18
1250	1.588	−101.377	22.21	3.560	−101.379	22.16	3.028	−101.379	22.18	3.335	−101.379	22.18

	Rest+This work			**Rest+Cube pruning**			**Rest+Gesmundo 1**			**Rest+Gesmundo 2**		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.301	−101.678	21.76	0.581	−101.598	22.13	0.559	−101.600	22.13	0.524	−101.601	22.13
10	0.309	−101.569	22.09	0.590	−101.541	22.25	0.606	−101.543	22.24	0.567	−101.545	22.23
15	0.310	−101.524	22.23	0.588	−101.513	22.26	0.549	−101.515	22.29	0.555	−101.517	22.28
20	0.305	−101.496	22.25	0.593	−101.496	22.26	0.535	−101.498	22.26	0.543	−101.499	22.23
25	0.342	−101.479	22.21	0.603	−101.483	22.20	0.608	−101.485	22.22	0.578	−101.486	22.22
50	0.345	−101.435	22.29	0.652	−101.447	22.24	0.600	−101.448	22.24	0.598	−101.449	22.23
75	0.398	−101.416	22.23	0.771	−101.431	22.22	0.643	−101.432	22.18	0.716	−101.432	22.18
100	0.400	−101.406	22.27	0.779	−101.419	22.21	0.737	−101.421	22.17	0.697	−101.422	22.21
150	0.479	−101.394	22.26	0.937	−101.408	22.29	0.790	−101.409	22.25	0.839	−101.408	22.25
200	0.564	−101.388	22.23	0.999	−101.400	22.28	0.905	−101.401	22.22	0.916	−101.401	22.22
300	0.645	−101.381	22.24	1.290	−101.392	22.26	1.086	−101.393	22.21	1.130	−101.392	22.19
400	0.786	−101.378	22.22	1.511	−101.387	22.21	1.394	−101.388	22.18	1.404	−101.387	22.16
500	0.892	−101.376	22.21	1.719	−101.383	22.19	1.524	−101.384	22.17	1.635	−101.383	22.17
750	1.243	−101.372	22.20	2.413	−101.377	22.17	1.993	−101.378	22.15	2.135	−101.377	22.17
1000	1.543	−101.370	22.18	2.950	−101.375	22.17	2.541	−101.375	22.17	2.701	−101.374	22.17

Table A.4: Numerical results for **cdec on hierarchical German–English** at select beam sizes.

	This work			Cube pruning			Additive cube		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.568	−13.662	14.48	1.331	−13.425	15.09	1.031	−13.818	14.18
10	1.301	−13.524	14.64	3.000	−13.343	15.18	2.186	−13.629	14.60
15	1.979	−13.454	14.80	5.145	−13.304	15.21	3.671	−13.546	14.67
20	2.745	−13.417	14.92	6.801	−13.281	15.20	4.694	−13.492	14.84
25	3.068	−13.389	14.96	7.928	−13.267	15.19	5.568	−13.454	14.85
50	4.740	−13.318	14.98	9.772	−13.228	15.24	6.926	−13.359	15.11
75	4.800	−13.286	15.09	11.875	−13.209	15.28	8.192	−13.314	15.09
100	5.736	−13.264	15.06	12.946	−13.198	15.18	9.184	−13.288	15.10
200	6.804	−13.224	15.06	14.804	−13.172	15.19	11.215	−13.238	15.05
300	7.177	−13.204	15.13	16.679	−13.161	15.17	11.429	−13.215	15.07
400	8.293	−13.191	15.07	17.101	−13.154	15.16	12.522	−13.201	15.07
500	8.076	−13.182	15.13	17.260	−13.149	15.14	12.494	−13.190	15.15
750	8.671	−13.169	15.14	20.917	−13.142	15.10	13.598	−13.174	15.16
1000	9.810	−13.160	15.09	20.206	−13.137	15.10	14.652	−13.164	15.15

	Rest+This work			Rest+Cube pruning			Rest+Additive cube		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.533	−13.538	14.72	1.292	−13.382	15.08	0.977	−13.741	14.39
10	1.515	−13.414	14.81	3.137	−13.303	15.16	2.399	−13.560	14.70
15	2.170	−13.360	14.88	5.240	−13.269	15.19	3.542	−13.477	14.75
20	3.153	−13.329	14.88	6.718	−13.249	15.21	4.652	−13.426	14.83
25	3.573	−13.307	14.88	8.158	−13.236	15.21	5.840	−13.393	14.84
75	5.309	−13.227	15.05	12.784	−13.185	15.19	8.547	−13.270	15.07
100	6.088	−13.209	15.08	13.880	−13.175	15.18	8.720	−13.248	15.01
150	7.173	−13.190	15.08	13.679	−13.162	15.16	10.206	−13.220	15.07
200	7.079	−13.179	15.03	15.638	−13.154	15.14	10.006	−13.204	15.02
300	7.942	−13.164	15.07	16.773	−13.145	15.13	11.850	−13.185	15.11
400	8.174	−13.155	15.07	17.088	−13.139	15.13	12.137	−13.174	15.11
500	8.639	−13.149	15.05	18.698	−13.135	15.13	12.327	−13.166	15.09
750	9.836	−13.139	15.06	19.663	−13.129	15.11	13.846	−13.152	15.11
1000	10.250	−13.133	15.06	19.814	−13.124	15.11	14.062	−13.143	15.12

Table A.5: Numerical results for **Moses on English–German with target syntax** at select beam sizes.

	This work			**Cube pruning**			**Additive cube**		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.393	−105.376	20.24	0.740	−105.124	20.89	0.784	−105.542	19.84
10	1.130	−104.963	20.62	2.035	−104.743	21.11	1.862	−105.032	20.46
15	1.957	−104.767	20.82	3.814	−104.611	21.18	3.292	−104.812	20.67
20	2.589	−104.495	20.97	4.704	−104.426	21.24	4.238	−104.561	20.76
25	2.782	−104.467	21.07	6.455	−104.381	21.29	4.975	−104.521	20.87
50	3.932	−104.397	21.23	8.864	−104.313	21.36	7.246	−104.428	21.10
75	5.332	−104.365	21.27	11.348	−104.297	21.34	8.638	−104.386	21.19
100	5.748	−104.344	21.29	12.822	−104.287	21.33	9.448	−104.361	21.28
150	7.105	−104.320	21.27	13.151	−104.274	21.31	10.331	−104.331	21.28
200	7.320	−104.308	21.28	15.943	−104.268	21.25	11.727	−104.314	21.33
300	8.619	−104.292	21.28	16.270	−104.259	21.32	11.941	−104.297	21.35
400	8.604	−104.283	21.35	16.791	−104.254	21.33	13.140	−104.286	21.37
500	10.138	−104.276	21.35	18.870	−104.251	21.32	13.656	−104.278	21.35
750	10.454	−104.265	21.38	20.952	−104.244	21.36	15.692	−104.267	21.33
1000	11.965	−104.260	21.37	22.495	−104.241	21.39	16.809	−104.260	21.33

	Rest+This work			**Rest+Cube pruning**			**Rest+Additive cube**		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
5	0.442	−105.093	20.92	0.802	−104.962	21.07	0.676	−105.306	20.49
10	1.150	−104.817	21.06	2.213	−104.721	21.26	1.829	−104.958	20.83
15	1.774	−104.668	21.16	3.735	−104.627	21.30	3.049	−104.779	21.00
20	2.516	−104.375	21.22	5.846	−104.316	21.33	4.204	−104.468	21.08
25	3.101	−104.354	21.20	6.027	−104.304	21.30	5.162	−104.434	21.10
50	3.988	−104.239	21.39	9.026	−104.239	21.38	6.997	−104.287	21.23
75	5.206	−104.215	21.39	10.954	−104.193	21.38	8.303	−104.252	21.34
100	5.562	−104.201	21.39	12.436	−104.183	21.40	9.377	−104.232	21.35
150	7.349	−104.182	21.43	13.672	−104.172	21.37	10.700	−104.207	21.42
200	7.597	−104.174	21.44	15.140	−104.165	21.35	12.160	−104.194	21.41
300	9.301	−104.163	21.46	16.490	−104.157	21.35	13.113	−104.178	21.38
400	10.081	−104.156	21.47	17.838	−104.152	21.37	13.353	−104.169	21.42
500	10.065	−104.152	21.47	18.585	−104.149	21.38	14.751	−104.163	21.41
750	11.112	−104.146	21.46	22.057	−104.143	21.43	15.454	−104.153	21.46
1000	11.838	−104.143	21.44	23.252	−104.140	21.42	16.940	−104.149	21.46

Table A.6: Numerical results for **Moses on German–English with target syntax** at select beam sizes.

	This work			Cube pruning			Gesmundo 1			Gesmundo 2		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
10	5.681	−228.482	26.73	8.829	−223.668	27.05	8.476	−223.740	27.04	8.351	−223.827	27.02
15	5.613	−226.395	26.97	9.165	−222.773	27.03	8.969	−222.834	27.03	8.932	−222.923	27.01
20	5.665	−225.169	27.10	10.540	−222.298	27.05	9.082	−222.370	27.05	9.533	−222.454	27.00
50	6.052	−222.419	27.21	14.636	−220.907	27.05	12.412	−220.968	27.04	12.655	−221.015	27.05
75	6.225	−221.538	27.14	16.795	−220.445	27.07	15.611	−220.484	27.08	14.912	−220.543	27.12
100	6.924	−221.040	27.14	18.644	−220.149	27.05	17.560	−220.195	27.06	17.524	−220.240	27.09
150	7.585	−220.479	27.18	25.498	−219.798	27.07	22.341	−219.839	27.11	22.575	−219.865	27.11
200	8.382	−220.117	27.16	32.301	−219.579	27.04	27.410	−219.636	27.09	27.525	−219.691	27.11
300	9.784	−219.733	27.10	41.168	−219.321	27.04	38.151	−219.347	27.06	38.882	−219.375	27.07

	Rest+This work			Rest+Cube pruning			Rest+Gesmundo 1			Rest+Gesmundo 2		
Beam	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**	**CPU**	**Model**	**BLEU**
10	5.369	−225.344	26.96	9.321	−222.868	26.97	8.543	−222.946	26.96	8.459	−222.977	26.97
15	5.806	−223.864	26.99	9.587	−222.111	26.88	9.064	−222.170	26.91	9.075	−222.208	26.93
20	5.970	−222.965	27.00	10.126	−221.652	26.90	9.496	−221.709	26.99	9.647	−221.777	26.99
50	6.031	−220.923	27.03	13.364	−220.516	26.95	12.518	−220.550	26.99	12.998	−220.637	26.96
75	6.399	−220.351	27.06	16.032	−220.072	26.98	14.762	−220.115	26.97	15.086	−220.186	26.98
100	7.064	−220.011	27.05	19.142	−219.860	26.98	17.466	−219.901	26.98	17.288	−219.936	26.99
150	7.855	−219.602	26.99	25.337	−219.552	26.93	22.820	−219.581	26.97	22.536	−219.626	26.99
200	8.738	−219.388	26.97	31.346	−219.394	26.95	27.879	−219.421	26.98	27.822	−219.448	26.97
300	10.360	−219.161	26.98	41.994	−219.163	26.97	38.306	−219.191	26.96	38.545	−219.200	26.96

Table A.7: Numerical results for **cdec on tree-to-tree French–English** at select beam sizes. Due to computational costs, cube pruning was run with beam sizes up to 300.