PBML

The Prague Bulletin of Mathematical Linguistics
NUMBER 93 JANUARY 2010 87–96

Fast and Extensible Phrase Scoring for Statistical Machine Translation

Christian Hardmeier

Fondazione Bruno Kessler, Trento, Italy

Abstract

Existing tools for generating phrase tables for phrase-based Statistical Machine Translation (SMT) are generally optimised towards low memory use to allow processing of large corpora with limited memory. Whilst being a reasonable design choice, this approach does not make optimal use of resources when the sufficient memory is available. We present *memscore*, a new open-source tool to score phrases in memory. Besides acting as a faster drop-in replacement for existing software, it implements a number of standard smoothing techniques and provides a platform for easy experimentation with new scoring methods.

1. Motivation

Phrase tables for Statistical Machine Translation (SMT) systems are commonly built from very large parallel corpora in order to obtain ample vocabulary coverage and sufficient quality of the translation probability estimates. On usual desktop computers, the size of the phrase tables extracted from a large corpus will often exceed the size of the physical memory available on the machine. Software tools used to estimate phrase tables from parallel corpora are designed to take this constraint into account. They do not try to load the complete data into memory at once. Instead, they process their inputs as data streams, relying on local information only, and make extensive use of temporary disk files and intermediate disk-bound sorting passes to access the right information at the right moment.

This approach ensures that the size of the parallel corpus that can be processed is limited only by the potentially very large amount of disk space available; the use of working memory is kept to a minimum. The result is almost unlimited scalability

Corresponding author: hardmeier@fbk.eu
Cite as: Christian Hardmeier. Fast and Extensible Phrase Scoring for Statistical Machine Translation. The Prague Bulletin of Mathematical Linguistics No. 93, 2010, pp. 87–96. ISBN 978-80-904175-4-0.
doi: 10.2478v10108-010-0007-5.

to very large corpora given sufficient disk space, but it seems wasteful not to exploit the available memory resources as fully as possible. Moreover, the stream-based approach to data processing makes it very expensive to access data that is not locally available, so the scoring functions that can be implemented are essentially limited to those that require only a small number of straight passes through the data. In recent years, random-access memory has become much cheaper, and the general availability of 64-bit computers has lifted another important restriction on memory size, such that academic sites now have access to computing equipment which can handle data sets in memory that were beyond reach even a few years ago. It is reasonable to use software that takes advantage of these new capabilities, not only to speed up SMT system training, but also to make it feasible to implement new phrase scoring algorithms that require more than just a few passes through the data.

Phrase-based Statistical Machine Translation (SMT) uses translation models in the form of phrase tables, in which phrase pairs consisting of a source language (SL) and a target language (TL) word sequence, s and t, are associated with a number of scores corresponding to different models of translation probabilities between s and t. Following Koehn et al. (2003), candidate phrase pairs are usually extracted from a parallel corpus with automatically generated word alignments. The forward and reverse conditional phrase translation probabilities $p(s|t)$ and $p(t|s)$ are then estimated by the relative frequency of a SL phrase in alignment with a given TL phrase and vice versa. To overcome the unreliability of these estimates for low-frequency phrases, phrase tables usually include maximum likelihood scores for both $p(s|t)$ and $p(t|s)$ as well as two additional lexical weight scores based on the word alignment probabilities of the individual component words of the source and the target phrases (Koehn et al., 2003).

The widely used *moses* toolkit for Statistical Machine Translation (Koehn et al., 2007) includes a tool called *phrase-extract* to extract phrase pairs from a word-aligned corpus and compute phrase translation probabilities and lexical weights. It is designed to process large amounts of corpus data on computers with relatively little random-access memory (RAM). To achieve this, the file system is used extensively to store temporary data. Phrases pairs are extracted from the parallel corpus and stored to disk. Scoring is done individually for the two forms of the conditional probability, $p(s|t)$ and $p(t|s)$, and for each scoring pass, the extracted phrases have to be sorted by target or source phrase, respectively. After scoring, the output of one of the two scoring runs is sorted again to match the order of the other run for merging. Another open-source implementation of SMT phrase scoring, Thot (Ortiz-Martínez et al., 2005), also works with temporary disk files to cut down on RAM usage.

In this paper, we present *memscore*, an open-source phrase scoring tool replicating and extending the functionality of the *score* component of the *phrase-extract* software bundled with *moses*. Like *score*, it takes as input a list of phrase pairs produced by the *extract* tool of *phrase-extract* and calculates phrase translation scores. Unlike *score*, it performs all the computations in RAM and does not require the input to be sorted in any way. As a result, scoring is much faster for data sets that fit completely in memory.

p(s|t)	p(t|s)	
`-s ml`	`-r ml`	maximum likelihood score
`-s wittenbell`	`-r wittenbell`	Witten-Bell smoothing
`-s absdiscount`	`-r absdiscount`	absolute discounting
`-s lexweights <file>`	`-r lexweights <file>`	lexical weights
`-s const <constant>`		constant phrase penalty

Table 1. memscore *command line options*

Its implementation as a C++ program designed with modularity in mind makes it easy to experiment with different scoring techniques. A small number of smoothing techniques are already implemented, and other methods can easily be added. The framework has also been used successfully for an experimental implementation of an iterative scoring algorithm. In the rest of the paper, we are going to describe the typical usage of *memscore*, some implementation details useful to those who want to implement their own scoring algorithms in the framework provided by the tool and a comparison of *memscore* with the *phrase-extract* scorer in terms of runtime performance on two common SMT tasks.

2. Usage

2.1. Invocation

The *memscore* tool takes as input a list of phrases extracted from a parallel corpus in the format used by the *phrase-extract* tool bundled with the *moses* decoder. The phrase list is read from standard input and does not need to be sorted. It prints to standard output a phrase table in the format used by *moses*.

The scores to be included in the phrase table are specified on the command line with the switches listed in table 1. Each score is selected by one of the command line options `-s` or `-r` followed by the identifier of the scorer. Additional arguments may follow if the scorer requires this. When the option `-s` is used to specify a scorer producing a conditional probability, the probability $p(s|t)$ is generated. Using the scoring option `-r` requests that the inverse probability $p(t|s)$ be output instead. Thus, to produce a phrase table with maximum likelihood probabilities and lexical weights in both directions and a constant phrase penalty, as typically created by the *moses* training scripts, you would use the following command line:

```
gzip -cd model/extract.gz |
   memscore  -s ml -s lexweights model/lex.e2f \
             -r ml -r lexweights model/lex.f2e \
             -s const 2.718 |
   gzip >model/phrase-table.gz
```

Here, the files `lex.e2f` and `lex.f2e` contain the lexical translation tables generated by the *moses* training script in training step 4, and `extract.gz` is the phrase extraction file produced in step 5. The *memscore* command itself replaces the scoring step 6. We plan to integrate this step smoothly into the standard *moses* training script, but at the time of writing, this has not been done yet.

In the configuration mentioned in the previous paragraph, the output of *memscore* is essentially identical to that of the reference implementation, *phrase-extract*. The estimates of the maximum-likelihood scores are exactly the same as those produced by *phrase-extract*. The lexical weights can be different if a certain phrase pair occurs in the input with more than one set of alignments. According to Koehn et al. (2003), the maximum score generated by any of the alignments should be used in this case. However, the reference implementation does not conform to this recommendation. Instead, it computes the lexical weight based on the alignment with the highest count in the input. If there are several alignments with equal counts, the one occurring earliest in the input stream is selected. Thus, the actual choice depends on the sorting order of the input. In our implementation, two different modes of operation are available: By default, *memscore* outputs the maximum lexical weight as suggested by Koehn et al. (2003). If the command line switch of the lexical weight scorer is given as `-s lexweights -AlignmentCount model/lex.e2f`, the lexical weight is based on the most frequent alignment instead. If there is a tie for the maximal count, the greatest score generated by any of the competing alignments is chosen. This mode of calculation matches the *phrase-extract* scoring more closely, but differences are still possible in a small number of cases.

2.2. File formats

The file formats processed by *memscore* are the same as those produced and used by the *moses* toolkit. They are illustrated in table 2. As input, a list of phrase pairs extracted from a parallel corpus is read. The three fields in each line, separated by the characters '␣||␣', are the source phrase, the target phrase and the word alignment. The alignment is specified as a list of alignment links between word numbers, where, e.g., a link `0-1` indicates that the first source word is aligned to the second target word. Each phrase pair should occur in the input as often as it can be extracted from the corpus. Sorting is not required. A suitable file is produced by the *moses* training procedure under the name `extract.gz`. The inverse extraction file `extract.inv.gz` is not needed when *memscore* is used.

The phrase table produced by *memscore* uses the same field delimiters. After the source and target phrases, the word alignment is given in a different format. For each source word, the third field contains a pair of parentheses with a comma-separated list of word indices in the target phrase aligned to this word. In the fourth field, there is a similar list for each target word. When a phrase pair occurs with different alignments in the input, the most frequent alignment is output. Ties are broken arbitrarily. The

Phrase extraction file (input):
```
gemäß ||| in accordance with ||| 0-0 0-1 0-2
```
Phrase table (output):
```
gemäß ||| in accordance with ||| (0,1,2) ||| (0) (0) (0) ||| s1 s2 ...
```
Lexical translation table:
```
gemäß accordance 0.0445155
```

Table 2. File record formats used by memscore

fifth field of the phrase table record contains the scores $s_1, s_2, \ldots$ as floating point numbers in the order in which the scorers were specified on the command line.

The lexical weight scorer additionally requires a list of lexical translation table as input. This table has records with three blank-separated fields giving the source word, the target word and the lexical translation score, which is estimated as the number of alignments between the source word and the target word in the corpus, divided by the number of occurrences of the target word. When the lexical weight scorer is used in reverse mode, the word translation probabilities must also be reversed. These are the files `lex.e2f` and `lex.f2e` provided by the *moses* training scripts.

3. Implementation

3.1. Architecture

The architecture of *memscore* has been designed to favour extensibility. Developers should be able to implement quickly new scoring mechanisms without having to spend time on parsing input files, designing compact data structures and dealing with memory management. The scoring code is cleanly separated from these ancillary functions. Also, computing the forward and reverse conditional probabilities $p(s|t)$ and $p(t|s)$ is handled transparently by *memscore*. The programmer only has to provide one implementation for the form $p(s|t)$; reverse scoring is available automatically.

The main components of *memscore* are outlined in table 3. The classes `PhraseTable`, `PhraseInfo` and `PhrasePairInfo` provide access the data structures in which information about the phrase table is stored. The algorithmic components are encapsulated in the subclasses of `PhraseStatistic`, which can be used to compute statistics about individual source language and target language phrases, and those of `PhraseScorer`, which represent the actual scoring algorithms, respectively.

The class `MemoryPhraseTable` takes care of parsing the input data and storing it in memory using a hash table provided by the C++ standard template library. The source and target phrases are stored in hash tables of their own and represented internally by numeric identifiers. For each phrase pair, a `PhrasePairInfo` data structure encapsulates the joint counts. For each SL or TL phrase, a `PhraseInfo` structure

Parent class	Derived classes	Description
PhraseTable	MemoryPhraseTable ReversePhraseTable	provide access to the phrase table
PhraseInfo		stores data about single phrases
PhrasePairInfo		stores data about phrase pairs
PhraseStatistic	PhraseLanguageModel ClosedPhraseLanguageModel	compute phrase-level statistics
PhraseScorer	MLPhraseScorer WittenBellPhraseScorer AbsoluteDiscountPhraseScorer ConstantPhraseScorer	implement phrase scoring algorithms

Table 3. Principal components of memscore

contains the marginal counts and the number of distinct phrases of the other language it is aligned with. Both the `PhraseInfo` and the `PhrasePairInfo` classes also have a mechanism by which more sophisticated scoring algorithms can request additional storage to be associated with phrases or phrase pairs. If, e. g., a phrase language model is used, it will ask for space to be reserved to cache the language model scores for each phrase. To avoid excessive memory consumption by features that are not actually used, the extra information for phrase pairs is stored in a variable-sized data structure that only includes the information actually used by the scoring algorithms selected by the user in a particular run. For the implementor of a scoring algorithm, this is handled transparently.

The class `ReversePhraseTable` is an adapter that provides access to the phrase table with the source and the target side exchanged. This makes it possible to use exactly the same implementations of any scorer for computing both $p(s|t)$ and $p(t|s)$.

Subclasses of `PhraseStatistic` calculate statistics of single SL or TL phrases, which can then be used by the actual scoring algorithm. This feature is not used by the standard scorers, all of which estimate scores based on phrase counts alone. However, an experimental scorer might also take into account other characteristics of the phrases. We provide two implementations of this interface: The `PhraseLanguageModel` class scores the phrases with an IRSTLM language model (Federico et al., 2008). The `ClosedLanguageModel` does the same, but normalises the language model scores over the phrases encountered in the phrase table, assuming a closed world of phrases enumerated by the input.

Finally, the subclasses of `PhraseScorer` do the actual scoring. At the moment, three algorithms with very simple implementations are available (table 4). The `MLPhraseScorer` computes the standard maximum likelihood estimate for a multinomial distribution based on relative frequencies. The `WittenBellPhraseScorer` uses

$$p(s|t) = \frac{c(s,t)}{c(t)} \qquad p(s|t) = \frac{c(s,t)}{c(t) + N_s(t)} \qquad p(s|t) = \frac{c(s,t) - \beta}{c(t)}$$

maximum likelihood | Witten-Bell | absolute discounting

$c(\cdot)$: (joint or marginal) counts β: discounting constant (see text)
$N_s(t)$: number of distinct s occurring with t

Table 4. Scoring methods implemented in memscore

the Witten-Bell estimate known from language modelling (Witten and Bell, 1991). Another estimate borrowed from language modelling (Ney et al., 1994) is calculated by the `AbsoluteDiscountPhraseScorer`, which reduces the joint count of each event by a discounting constant $\beta = n_1/(n_1 + 2n_2)$, where n_1 and n_2 are the number of phrase pairs occurring exactly once or twice in the parallel corpus, respectively. In both cases, the probability mass is not redistributed to any backoff distributions, so the probabilities will not sum to 1 over the closed world of the phrase table. This type of smoothing avoids the typical overconfident estimates for phrase pairs with low counts that maximum likelihood estimation is subject to.

3.2. Memory management

The operation of *memscore* can be divided in three phases. First, the input data is loaded and the internal data structures are constructed. Next, the `PhraseScorer` classes have the opportunity to collect any statistics they require by accessing the data in an arbitrary way. The maximum likelihood and Witten-Bell scorers do nothing in this stage; the absolute discounting scorer computes its discounting constant β. Finally, the scorers are requested to emit their score estimates in a final, ordered pass through all the phrase pairs.

In terms of memory consumption, the first stage is characterised by the allocation of a very large amount of memory for a multitude of small objects representing phrase or phrase pair properties. In the next two phases, memory usage remains essentially constant; all the memory is freed at once on program termination. The default memory allocators do not cope well with this usage pattern. Memory and execution time profiling revealed that the excessive allocation of small objects leads to overhead memory consumption of up to one third of the total amount of space requested because the memory allocator associates a certain amount of accounting information with each memory block allocated, and the time wasted on memory allocation and deallocation far exceeds the time spent on scoring. To overcome these problems, considerable effort went into the optimisation of memory allocation patterns. In some important cases, allocation and freeing of large numbers of small objects was made considerably more efficient by judicious use of the memory pools provided by the

	Europarl DE-EN		**NIST** AR-EN	
Corpus size (sentences)	1,252,747		4,654,686	
Corpus size (English tokens)	34,731,010		147,135,694	
Phrase pairs (types)	28,251,755		115,474,492	
Phrase pairs (instances)	64,271,574		318,961,124	
	score	*memscore*	*score*	*memscore*
Time (h : mm)	1 : 05	0 : 26	5 : 46	2 : 14
Peak memory usage	12.3 GB	15.7 GB	5 GB	62.2 GB
Single iteration		13.2 s		38.6 s

Table 5. Scorer performance on Europarl and NIST tasks

Boost library, which allocate single large pools of memory to hold many small objects of equal size. In this way, the memory management overhead could be significantly reduced both in terms of time and space, so that under normal conditions most of the delays now stem from input/output operations, not from memory management.

4. Performance

To evaluate the performance of *memscore* relative to *phrase-extract*, we tested it on two tasks of different sizes. As a medium size task, we trained a phrase table on the German-English portion of the Europarl corpus (Koehn, 2005). The large task uses all parallel data of the Arabic-English constrained data set of the 2009 NIST Machine Translation evaluation campaign. The experiments were performed on the computing cluster of Fondazione Bruno Kessler (FBK-irst), Trento, on Linux computers with 2.5 GHz Intel Xeon CPUs. The cluster setup at FBK also defined the constraints for the practical usability of the scoring tool and for the comparison with *phrase-extract*. The maximum amount of random-access memory on a single machine is 70 GB. The local disks of the computing nodes are relatively small, so that all data must be read from and written to a network-mounted drive, which has a significant negative impact on the performance of both *memscore* and *phrase-extract*. Temporary files created during the initial and intermediate sorting steps of the *phrase-extract* procedure were kept on the local disk. The *memscore* procedure did not require any sorting steps.

The results of the experiments can be found in table 5. On the cluster hardware at FBK, it is possible to train even a NIST system in memory using *memscore*, even though this is clearly pushing it to the limit. The memory usage of *memscore* is approximately proportional to the size of the phrase table, which in turn depends on the corpus size. In this way, the maximum corpus size that can be handled on a machine with a given

amount of memory can be estimated. The memory consumption of the scoring procedure with *phrase-extract*, on the other hand, is clearly not correlated with the corpus size; indeed, in our example runs it was greater for the smaller corpus. The *score* program itself consumes hardly any memory except for storing the lexical translation table. It is likely that the peak values reported in table 5 are due to the GNU *sort* utility which for some reason settled on a different trade-off between using temporary disk files and increased resident memory usage in the two conditions.

The time required to estimate a phrase table is roughly halved by the use of *memscore*. This time is largely dominated by network input/output operations, and the difference roughly reflects the fact that *phrase-extract* scores the two phrase table halves separately, whereas *memscore* can do it in one step. It should also be noted that, as a result of being I/O-dominated, the timing is very sensitive to the overall load on the machines and the network, a factor not controlled in the experiments, so the indications should be taken with a grain of salt. Experience shows that the actual scoring is very fast compared to loading and saving the data, so it is possible to apply iterative scoring methods even for large data sets without incurring a noticeable performance penalty.

To illustrate this effect, we ran another experiment to determine the cost of a single iteration through the complete phrase table excluding the time to load and save the table. We simulated a simple iterative scoring algorithm performing 200 passes through the whole data. In each pass, an operation identical in cost to a relative frequency computation, composed of looking up the marginal count in the phrase information structure and a division, was executed for every phrase pair. The last row in table 5 reports the average time per iteration, which gives an estimate of the marginal cost of an additional pass through the data in an iterative algorithm once the loading and saving times are accounted for.

5. Future work

In its current state, *memscore* can be a useful tool to speed up the training pipeline of an SMT system when computers with large amounts of random-access memory are available. Its extensible design also makes it easy to implement and test new scoring methods. We hope that the public availability of an extensible scoring framework will reduces the work involved in publishing new scoring methods in the form of ready-to-use implementations.

So far, we have been concentrating on implementing the phrase scoring algorithm, relying on the *moses* training scripts to extract phrases from the word-aligned parallel corpus and to estimate the word-to-word translation probabilities used in calculating lexical weights. It should be relatively straightforward, however, to integrate these steps directly into *memscore*. The scoring tool would build its internal representations directly from the parallel corpus. Phrase extraction files and word-to-word dictionaries would be saved to disk only on request. In addition to making *memscore* more

self-contained, this could also lead to a considerable reduction in the total amount of disk space required, on the one hand, and of disk input/output activity, on the other hand. In a networked environment, where data resides on remote disks, loading only the aligned parallel corpus rather than loading and storing large phrase extraction files could speed up the training process even further.

Another sorting step in the training pipeline could be avoided by making *memscore* output the phrase table in the order required by *processPhraseTable*, which creates binary phrase tables to be used with *moses*. Since *memscore* internally stores the phrase pairs in a hash table, which naturally iterates over its elements in a well-defined order, this only requires defining suitable comparison operators for the phrase representation based on numerical identifiers used internally.

Finally, *memscore* could be extended to estimate lexical reordering tables, so that it would cover the complete training of a phrase-based SMT system given the word alignments.

Acknowledgements

This work was supported by the EuroMatrixPlus project (IST-231720), which is funded by the European Commission under the Seventh Framework Programme for Research and Technological Development. I am indebted to Gabriele Musillo and Marcello Federico for their valuable comments on this paper.

Bibliography

Federico, Marcello, Nicola Bertoldi, and Mauro Cettolo. Irstlm: an open source toolkit for handling large scale language models. In *Proceedings of Interspeech*, Brisbane, 2008.

Koehn, Philipp. Europarl: a corpus for statistical machine translation. In *Proceedings of MT Summit X*, pages 79–86, Phuket, Thailand, 2005. AAMT.

Koehn, Philipp, Franz Josef Och, and Daniel Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 conference of the North American chapter of the Association for Computational Linguistics on Human Language Technology*, pages 48–54, Edmonton, 2003.

Koehn, Philipp et al. Moses: open source toolkit for statistical machine translation. In *Annual meeting of the Association for Computational Linguistics: Demonstration session*, pages 177–180, Prague, 2007.

Ney, Hermann, Ute Essen, and Reinhard Kneser. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech and Language*, 8:1–38, 1994.

Ortiz-Martínez, D., I. García-Varea, and F. Casacuberta. Thot: a toolkit to train phrase-based statistical translation models. In *Proceedings of MT Summit X*, pages 141–148, Phuket, Thailand, 2005. AAMT.

Witten, Ian H. and Timothy C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4): 1085–1094, 1991.