Hierarchical Phrase-Based Translation with Suffix Arrays

Adam Lopez
Computer Science Department
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742 USA
alopez@cs.umd.edu

Abstract

A major engineering challenge in statistical machine translation systems is the efficient representation of extremely large translation rulesets. In phrase-based models, this problem can be addressed by storing the training data in memory and using a suffix array as an efficient index to quickly lookup and extract rules on the fly. *Hierarchical* phrase-based translation introduces the added wrinkle of source phrases with gaps. Lookup algorithms used for contiguous phrases no longer apply and the best approximate pattern matching algorithms are much too slow, taking several minutes per sentence. We describe new lookup algorithms for hierarchical phrase-based translation that reduce the empirical computation time by nearly two orders of magnitude, making on-the-fly lookup feasible for source phrases with gaps.

1 Introduction

Current statistical machine translation systems rely on very large rule sets. In phrase-based systems, rules are extracted from parallel corpora containing tens or hundreds of millions of words. This can result in millions of rules using even the most conservative extraction heuristics. Efficient algorithms for rule storage and access are necessary for practical decoding algorithms. They are crucial to keeping up with the ever-increasing size of parallel corpora, as well as the introduction of new data sources such as web-mined and comparable corpora.

Until recently, most approaches to this problem involved substantial tradeoffs. The common practice of test set filtering renders systems impractical for all but batch processing. Tight restrictions on phrase length curtail the power of phrase-based models. However, some promising engineering solutions are emerging. Zens and Ney (2007) use a disk-based prefix tree, enabling efficient access to phrase tables much too large to fit in main memory. An alternative approach introduced independently by both Callison-Burch et al. (2005) and Zhang and Vogel (2005) is to store the training data itself in memory, and use a *suffix array* as an efficient index to look up, extract, and score phrase pairs on the fly. We believe that the latter approach has several important applications (§7).

So far, these techniques have focused on phrase-based models using *contiguous* phrases (Koehn et al., 2003; Och and Ney, 2004). Some recent models permit *discontiguous* phrases (Chiang, 2007; Quirk et al., 2005; Simard et al., 2005). Of particular interest to us is the hierarchical phrase-based model of Chiang (2007), which has been shown to be superior to phrase-based models. The ruleset extracted by this model is a superset of the ruleset in an equivalent phrase-based model, and it is an order of magnitude larger. This makes efficient rule representation even more critical. We tackle the problem using the online rule extraction method of Callison-Burch et al. (2005) and Zhang and Vogel (2005).

The problem statement for our work is: *Given an input sentence, efficiently find all hierarchical phrase-based translation rules for that sentence in the training corpus.*

Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pp. 976–985, Prague, June 2007.

We first review suffix arrays (§2) and hierarchical phrase-based translation (§3). We show that the obvious approach using state-of-the-art pattern matching algorithms is hopelessly inefficient (§4). We then describe a series of algorithms to address this inefficiency (§5). Our algorithms reduce computation time by two orders of magnitude, making the approach feasible (§6). We close with a discussion that describes several applications of our work (§7).

2 Suffix Arrays

A suffix array is a data structure representing all suffixes of a corpus in lexicographical order (Manber and Myers, 1993). Formally, for a text T, the ith suffix of T is the substring of the text beginning at position i and continuing to the end of T. This suffix can be uniquely identified by the index i of its first word. The suffix array SA_T of T is a permutation of $[1, |T|]$ arranged by the lexicographical order of the corresponding suffixes. This representation enables fast lookup of any contiguous substring using binary search. Specifically, all occurrences of a length-m substring can be found in $O(m + \log |T|)$ time (Manber and Myers, 1993). [1]

Callison-Burch et al. (2005) and Zhang and Vogel (2005) use suffix arrays as follows.

1. Load the source training text F, the suffix array SA_F, the target training text E, and the alignment A into memory.

2. For each input sentence, look up each substring (phrase) $\bar{f}$ of the sentence in the suffix array.

3. For each instance of $\bar{f}$ found in F, find its aligned phrase $\bar{e}$ using the phrase extraction method of Koehn et al. (2003).

4. Compute the relative frequency score $p(\bar{e}|\bar{f})$ of each pair using the count of the extracted pair and the marginal count of $\bar{f}$.

5. Compute the lexical weighting score of the phrase pair using the alignment that gives the best score.

6. Use the scored rules to translate the input sentence with a standard decoding algorithm.

A difficulty with this approach is step 3, which can be quite slow. Its complexity is linear in the number of occurrences of the source phrase $\bar{f}$. Both Callison-Burch et al. (2005) and Zhang and Vogel (2005) solve this with sampling. If a source phrase appears more than k times, they sample only k occurrences for rule extraction. Both papers report that translation performance is nearly identical to extracting all possible phrases when $k = 100$. [2]

3 Hierarchical Phrase-Based Translation

We consider the hierarchical translation model of Chiang (2007). Formally, this model is a synchronous context-free grammar. The lexicalized translation rules of the grammar may contain a single nonterminal symbol, denoted X. We will use a, b, c and d to denote terminal symbols, and u, v, and w to denote (possibly empty) sequences of these terminals. We will additionally use α and β to denote (possibly empty) sequences containing both terminals and nonterminals.

A translation rule is written $X \rightarrow \alpha/\beta$. This rule states that a span of the input matching α is replaced by β in translation. We require that α and β contain an equal number (possibly zero) of *coindexed* nonterminals. An example rule with coindexes is $X \rightarrow uX_{\boxed{1}}vX_{\boxed{2}}w/u'X_{\boxed{2}}v'X_{\boxed{1}}w'$. When discussing only the source side of such rules, we will leave out the coindexes. For instance, the source side of the above rule will be written $uXvXw$. [3]

For the purposes of this paper, we adhere to the restrictions described by Chiang (2007) for rules extracted from the training data.

- Rules can contain at most two nonterminals.

- Rules can contain at most five terminals.

- Rules can span at most ten words.

[1] Abouelhoda et al. (2004) show that lookup can be done in optimal $O(m)$ time using some auxiliaray data structures. For our purposes $O(m + \log |T|)$ is practical, since for the 27M-word corpus used to carry out our experiments, $\log |T| \sim 25$.

[2] A sample size of 100 is actually quite small for many phrases, some of which occur tens or hundreds of thousands of times. It is perhaps surprising that such a small sample size works as well as the full data. However, recent work by Och (2005) and Federico and Bertoldi (2006) has shown that the statistics used by phrase-based systems are not very precise.

[3] In the canonical representation of the grammar, source-side coindexes are always in sorted order, making them unambiguous.

- Nonterminals must span at least two words.
- Adjacent nonterminals are disallowed in the source side of a rule.

Expressed more economically, we say that our goal is to search for source phrases in the form u, uXv, or $uXvXw$, where $1 \le |uvw| \le 5$, and $|v| > 0$ in the final case. Note that the model also allows rules in the form Xu, uX, XuX, $XuXv$, and $uXvX$. However, these rules are lexically identical to other rules, and thus will match the same locations in the source text.

4 The Collocation Problem

On-the-fly lookup using suffix arrays involves an added complication when the rules are in form uXv or $uXvXw$. Binary search enables fast lookup of contiguous substrings. However, it cannot be used for discontiguous substrings. Consider the rule $aXbXc$. If we search for this rule in the following logical suffix array fragment, we will find the bold-faced matches.

...										
a	c	a	c	**b**	a	d	**c**	a	d	...
a	c	a	d	b	a	a	d	b	d	...
a	d	d	**b**	a	a	d	a	b	**c**	...
a	d	d	b	d	a	a	b	b	a	...
a	d	d	**b**	d	d	**c**	a	a	a	...
...										

Even though these suffixes are in lexicographical order, matching suffixes are interspersed with non-matching suffixes. We will need another algorithm to find the source rules containing at least one X surrounded by nonempty sequences of terminal symbols.

4.1 Baseline Approach

In the pattern-matching literature, words spanned by the nonterminal symbols of Chiang's grammar are called *don't cares* and a nonterminal symbol in a query pattern that matches a sequence of don't cares is called a *variable length gap*. The search problem for patterns containing these gaps is a variant of *approximate pattern matching*, which has received substantial attention (Navarro, 2001). The best algorithm for pattern matching with variable-length gaps in a suffix array is a recent algorithm by Rahman et al. (2006). It works on a pattern $w_1Xw_2X...w_I$ consisting of I contiguous substrings $w_1, w_2, ...w_I$, each separated by a gap. The algorithm is straightforward. After identifying all n_i occurrences of each w_i in $O(|w_i| + \log|T|)$ time, collocations that meet the gap constraints are computed using an efficient data structure called a *stratified tree* (van Emde Boas et al., 1977). [4] Although we refer the reader to the source text for a full description of this data structure, its salient characteristic is that it implements priority queue operations *insert* and *next-element* in $O(\log\log|T|)$ time. Therefore, the total running time for an algorithm to find all contiguous subpatterns and compute their collocations is $O(\sum_{i=1}^{I}[|w_i| + log|T| + n_i \log\log|T|])$.

We can improve on the algorithm of Rahman et al. (2006) using a variation on the idea of hashing. We exploit the fact that our large text is actually a collection of relatively short sentences, and that collocated patterns must occur in the same sentence in order to be considered a rule. Therefore, we can use the sentence id of each subpattern occurrence as a kind of hash key. We create a hash table whose size is exactly the number of sentences in our training corpus. Each location of the partially matched pattern $w_1X...Xw_i$ is inserted into the hash bucket with the matching sentence id. To find collocated patterns w_{i+1}, we probe the hash table with each of the n_{i+1} locations for that subpattern. When a match is found, we compare the element with all elements in the bucket to see if it is within the window imposed by the phrase length constraints. Theoretically, the worst case for this algorithm occurs when all elements of both sets resolve to the same hash bucket, and we must compare all elements of one set with all elements of the other set. This leads to a worst case complexity of $O(\sum_{i=1}^{I}[|w_i| + log|T|] + \prod_{i=1}^{I} n_i)$. However, for real language data the performance for sets of any significant size will be $O(\sum_{i=1}^{I}[|w_i| + log|T| + n_i])$, since most patterns will occur once in any given sentence.

4.2 Analysis

It is instructive to compare this with the complexity for contiguous phrases. In that case, total lookup time is $O(|w| + log|T|)$ for a contiguous pattern w.

[4] Often known in the literature as a *van Emde Boas tree* or *van Emde Boas priority queue*.

The crucial difference between the contiguous and discontiguous case is the added term $\sum_{i=1}^{I} n_i$. For even moderately frequent subpatterns this term dominates complexity.

To make matters concrete, consider the training corpus used in our experiments (§6), which contains 27M source words. The three most frequent unigrams occur 1.48M, 1.16M and 688K times – the first two occur on average more than once per sentence. In the worst case, looking up a contiguous phrase containing any number and combination of these unigrams requires no more than 25 comparison operations. In contrast, the worst case scenario for a pattern with a single gap, bookended on either side by the most frequent word, requires over *two million* operations using our baseline algorithm and over *thirteen million* using the algorithm of Rahman et al. (2006). A single frequent word in an input sentence is enough to cause noticeable slowdowns, since it can appear in up to 530 hierarchical rules.

To analyze the cost empirically, we ran our baseline algorithm on the first 50 sentences of the NIST Chinese-English 2003 test set and measured the CPU time taken to compute collocations. We found that, on average, it took 2241.25 seconds (~37 minutes) *per sentence* just to compute all of the needed collocations. By comparison, decoding time per sentence is roughly 10 seconds with moderately aggressive pruning, using the Python implementation of Chiang (2007).

5 Solving the Collocation Problem

Clearly, looking up patterns in this way is not practical. To analyze the problem, we measured the amount of CPU time per computation. Cumulative lookup time was dominated by a very small fraction of the computations (Fig. 1). As expected, further analysis showed that these expensive computations all involved one or more very frequent subpatterns. In the worst cases a single collocation took several seconds to compute. However, there is a silver lining. Patterns follow a Zipf distribution, so the number of pattern *types* that cause the problem is actually quite small. The vast majority of patterns are rare. Therefore, our solution focuses on computations where one or more of the component patterns is frequent. Assume that we are computing a collocation of pattern $w_1 X ... X w_i$ and pattern w_{i+1}, and we know all locations of each. There are three cases.

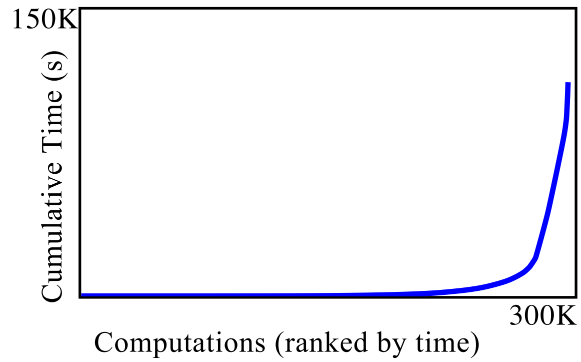

Figure 1: Ranked computations vs. cumulative time. A small fraction of all computations account for most of the computational time.

- If both patterns are frequent, we resort to a precomputed intersection (§5.1). We were not aware of any algorithms to substantially improve the efficiency of this computation when it is requested on the fly, but precomputation can be done in a single pass over the text at decoder startup.
- If one pattern is frequent and the other is rare, we use an algorithm whose complexity is dependent mainly on the frequency of the rare pattern (§5.2). It can also be used for pairs of rare patterns when one pattern is much rarer than the other.
- If both patterns are rare, no special algorithms are needed. Any linear algorithm will suffice. However, for reasons described in §5.3, our other collocation algorithms depend on sorted sets, so we use a merge algorithm.

Finally, in order to cut down on the number of unnecessary computations, we use an efficient method to enumerate the phrases to lookup (§5.4). This method also forms the basis of various caching strategies for additional speedups. We analyze the memory use of our algorithms in §5.5.

5.1 Precomputation

Precomputation of the most expensive collocations can be done in a single pass over the text. As input, our algorithm requires the identities of the k

most frequent contiguous patterns. [5] It then iterates over the corpus. Whenever a pattern from the list is seen, we push a tuple consisting of its identity and current location onto a queue. Whenever the oldest item on the queue falls outside the maximum phrase length window with respect to the current position, we compute that item's collocation with all succeeding patterns (subject to pattern length constraints) and pop it from the queue. We repeat this step for every item that falls outside the window. At the end of each sentence, we compute collocations for any remaining items in the queue and then empty it.

Our precomputation includes the most frequent n-gram subpatterns. Most of these are unigrams, but in our experiments we found 5-grams among the 1000 most frequent patterns. We precompute the locations of source phrase uXv for any pair u and v that both appear on this list. There is also a small number of patterns uXv that are very frequent. We cannot easily obtain a list of these in advance, but we observe that they always consist of a pair u and v of patterns from near the top of the frequency list. Therefore we also precompute the locations $uXvXw$ of patterns in which both u and v are among these *super-frequent* patterns (all unigrams), treating this as the collocation of the frequent pattern uXv and frequent pattern w. We also compute the analagous case for u and vXw.

5.2 Fast Intersection

For collocations of frequent and rare patterns, we use a fast set intersection method for sorted sets called double binary search (Baeza-Yates, 2004). [6] It is based on the intuition that if one set in a pair of sorted sets is much smaller than the other, then we can compute their intersection efficiently by performing a binary search in the larger *data set* D for each element of the smaller *query set* Q.

Double binary search takes this idea a step further. It performs a binary search in D for the median element of Q. Whether or not the element is found, the search divides both sets into two pairs of smaller sets that can be processed recursively. Detailed analysis and empirical results on an information retrieval task are reported in Baeza-Yates (2004) and Baeza-Yates and Salinger (2005). If $|Q| \log |D| < |D|$ then the performance is guaranteed to be sublinear. In practice it is often sublinear even if $|Q| \log |D|$ is somewhat larger than $|D|$. In our implementation we simply check for the condition $\lambda |Q| \log |D| < |D|$ to decide whether we should use double binary search or the merge algorithm. This check is applied in the recursive cases as well as for the initial inputs. The variable λ can be adjusted for performance. We determined experimentally that a good value for this parameter is 0.3.

5.3 Obtaining Sorted Sets

Double binary search requires that its input sets be in sorted order. However, the suffix array returns matchings in lexicographical order, not numeric order. The algorithm of Rahman et al. (2006) deals with this problem by inserting the unordered items into a stratified tree. This requires $O(n \log \log |T|)$ time for n items. If we used the same strategy, our algorithm would no longer be sublinear.

An alternative is to precompute all n-gram occurrences in order and store them in an inverted index. This can be done in one pass over the data. [7] This approach requires a separate inverted index for each n, up to the maximum n used by the model. The memory cost is one length-$|T|$ array per index.

In order to avoid the full $n|T|$ cost in memory, our implementation uses a mixed strategy. We keep a precomputed inverted index only for unigrams. For bigrams and larger n-grams, we generate the index on the fly using stratified trees. This results in a superlinear algorithm for intersection. However, we can exploit the fact that we must compute collocations multiple times for each input n-gram by caching the sorted set after we create it (The caching strategy is described in §5.4). Subsequent computations involving this n-gram can then be done in linear or sublinear time. Therefore, the cost of building the inverted index on the fly is amortized over a large number of computations.

[5] These can be identified using a single traversal over a *longest common prefix (LCP) array*, an auxiliary data structure of the suffix array, described by Manber and Myers (1993). Since we don't need the LCP array at runtime, we chose to do this computation once offline.

[6] Minor modifications are required since we are computing collocation rather than intersection. Due to space constraints, details and proof of correctness are available in Lopez (2007a).

[7] We combine this step with the other precomputations that require a pass over the data, thereby removing a redundant $O(|T|)$ term from the startup cost.

5.4 Efficient Enumeration

A major difference between contiguous phrase-based models and hierarchical phrase-based models is the number of rules that potentially apply to an input sentence. To make this concrete, on our data, with an average of 29 words per sentence, there were on average 133 contiguous phrases of length 5 or less that applied. By comparison, there were on average 7557 hierarchical phrases containing up to 5 words. These patterns are obviously highly overlapping and we employ an algorithm to exploit this fact. We first describe a baseline algorithm used for contiguous phrases (§5.4.1). We then introduce some improvements (§5.4.2) and describe a data structure used by the algorithm (§5.4.3). Finally, we discuss some special cases for discontiguous phrases (§5.4.4).

5.4.1 The Zhang-Vogel Algorithm

Zhang and Vogel (2005) present a clever algorithm for contiguous phrase searches in a suffix array. It exploits the fact that for each m-length source phrase that we want to look up, we will also want to look up its $(m-1)$-length prefix. They observe that the region of the suffix array containing all suffixes prefixed by ua is a subset of the region containing the suffixes prefixed by u. Therefore, if we enumerate the phrases of our sentence in such a way that we always search for u before searching for ua, we can restrict the binary search for ua to the range containing the suffixes prefixed by u. If the search for u fails, we do not need to search for ua at all. They show that this approach leads to some time savings for phrase search, although the gains are relatively modest since the search for contiguous phrases is not very expensive to begin with. However, the potential savings in the discontiguous case are much greater.

5.4.2 Improvements and Extensions

We can improve on the Zhang-Vogel algorithm. An m-length contiguous phrase aub depends not only on the existence of its prefix au, but also on the existence of its suffix ub. In the contiguous case, we cannot use this information to restrict the starting range of the binary search, but we can check for the existence of ub to decide whether we even need to search for aub at all. This can help us avoid searches that are guaranteed to be fruitless.

Now consider the discontiguous case. As in the analogous contiguous case, a phrase $a\alpha b$ will only exist in the text if its *maximal prefix* $a\alpha$ and *maximal suffix* αb both exist in the corpus and overlap at specific positions. [8] Searching for $a\alpha b$ is potentially very expensive, so we put all available information to work. Before searching, we require that both $a\alpha$ and αb exist. Additionally, we compute the location of $a\alpha b$ using the locations of both maximal subphrases. To see why the latter optimization is useful, consider a phrase $abXcd$. In our baseline algorithm, we would search for ab and cd, and then perform a computation to see whether these subphrases were collocated within an elastic window. However, if we instead use $abXc$ and $bXcd$ as the basis of the computation, we gain two advantages. First, the number elements of each set is likely to be smaller then in the former case. Second, the computation becomes simpler, because we now only need to check to see whether the patterns exactly overlap with a starting offset of one, rather than checking within a window of locations.

We can improve efficiency even further if we consider cases where the same substring occurs more than once within the same sentence, or even in multiple sentences. If the computation required to look up a phrase is expensive, we would like to perform the lookup only once. This requires some mechanism for caching. Depending on the situation, we might want to cache only certain subsets of phrases, based on their frequency or difficulty to compute. We would also like the flexibility to combine on-the-fly lookups with a partially precomputed phrase table, as in the online/offline mixture of Zhang and Vogel (2005).

We need a data structure that provides this flexibility, in addition to providing fast access to both the maximal prefix and maximal suffix of any phrase that we might consider.

5.4.3 Prefix Trees and Suffix Links

Our search optimizations are easily captured in a *prefix tree* data structure augmented with *suffix links*. Formally, a prefix tree is an unminimized deterministic finite-state automaton that recognizes all of the patterns in some set. Each node in the tree repre-

[8] Except when $\alpha = X$, in which case a and b must be collocated within a window defined by the phrase length constraints.

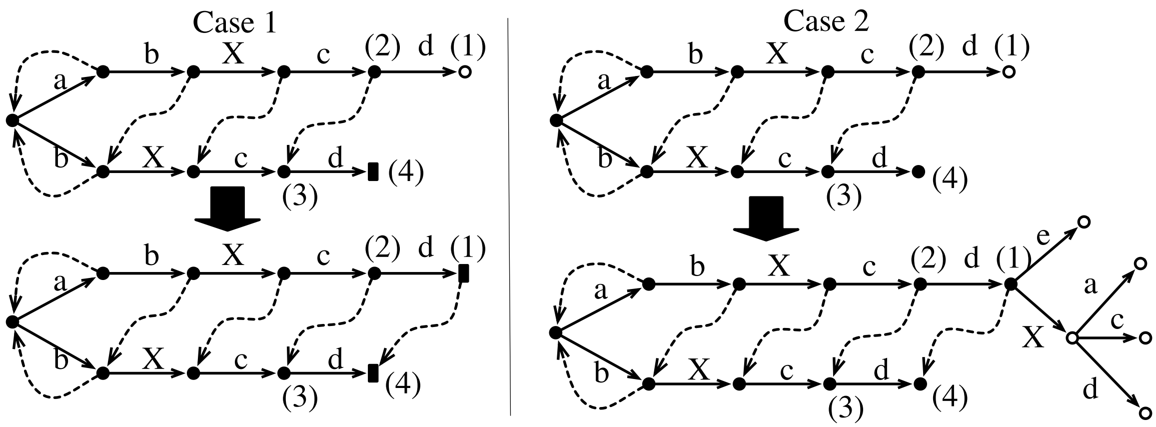

Figure 2: Illustration of prefix tree construction showing a partial prefix tree, including suffix links. Suppose we are interested in pattern $abXcd$, represented by node (1). Its prefix is represented by node (2), and node (2)'s suffix is represented by node (3). Therefore, node (1)'s suffix is represented by the node pointed to by the d-edge from node (3), which is node (4). There are two cases. In case 1, node (4) is inactive, so we can mark node (1) inactive and stop. In case 2, node (4) is active, so we compute the collocation of $abXc$ and $bXcd$ with information stored at nodes (2) and (4), using either a precomputed intersection, double binary search, or merge, depending on the size of the sets. If the result is empty, we mark the node inactive. Otherwise, we store the results at node (1) and add its successor patterns to the frontier for the next iteration. This includes all patterns containing exactly one more terminal symbol than the current pattern.

sents the prefix of a unique pattern from the set that is specified by the concatenation of the edge labels along the path from the root to that node. A suffix link is a pointer from a node representing path $a\alpha$ to the node representing path α. We will use this data structure to record the set of patterns that we have searched for and to cache information for those that were found successfully.

Our algorithm generates the tree breadth-search along a *frontier*. In the mth iteration we only search for patterns containing m terminal symbols. Regardless of whether we find a particular pattern, we create a node for it in the tree. If the pattern was found in the corpus, its node is marked *active*. Otherwise, it is marked *inactive*. For found patterns, we store either the endpoints of the suffix array range containing the phrase (if it is contiguous), or the list of locations at which the phrase is found (if it is discontiguous). We can also store the extracted rules. [9] Whenever a pattern is successfully found, we add all patterns with $m+1$ terminals that are prefixed by it to the frontier for processing in the next iteration.

To search for a pattern, we use location information from its parent node, which represents its maximal prefix. Assuming that the node represents phrase αb, we find the node representing its maximal suffix by following the b-edge from the node pointed to by its parent node's suffix link. If the node pointed to by this suffix link is inactive, we can mark the node inactive without running a search. When a node is marked inactive, we discontinue search for phrases that are prefixed by the path it represents. The algorithm is illustrated in Figure 2.

5.4.4 Special Cases for Phrases with Gaps

A few subtleties arise in the extraction of hierarchical patterns. Gaps are allowed to occur at the beginning or end of a phrase. For instance, we may have a source phrase Xu or uX or even XuX. Although each of these phrases requires its own path in the prefix tree, they are lexically identical to phrase u. An analogous situation occurs with the patterns $XuXv$, $uXvX$, and uXv. There are two cases that we are concerned with.

The first case consists of all patterns prefixed with X. The paths to nodes representing these patterns

[9] Conveniently, the implementation of Chiang (2007) uses a prefix tree grammar encoding, as described in Klein and Manning (2001). Our implementation decorates this tree with additional information required by our algorithms.

will all contain the X-edge originating at the root node. All of these paths form the *shadow subtree*. Path construction in this subtree proceeds differently. Because they are lexically identical to their suffixes, they are automatically extended if their suffix paths are active, and they inherit location information of their suffixes.

The second case consists of all patterns suffixed with X. Whenever we successfully find a new pattern α, we automatically extend it with an X edge, provided that αX is allowed by the model constraints. The node pointed to by this edge inherits its location information from its parent node (representing the maximal prefix α).

Note that both special cases occur for patterns in the form XuX.

5.5 Memory Requirements

As shown in Callison-Burch et al. (2005), we must keep an array for the source text F, its suffix array, the target text E, and alignment A in memory. Assuming that A and E are roughly the size of F, the cost is $4|T|$. If we assume that all data use vocabularies that can be represented using 32-bit integers, then our 27M word corpus can easily be represented in around 500MB of memory. Adding the inverted index for unigrams increases this by 20%. The main additional cost in memory comes from the storage of the precomputed collocations. This is dependent both on the corpus size and the number of collocations that we choose to precompute. Using detailed timing data from our experiments we were able to simulate the memory-speed tradeoff (Fig. 3). If we include a trigram model trained on our bitext and the Chinese Gigaword corpus, the overall storage costs for our system are approximately 2GB.

6 Experiments

All of our experiments were performed on Chinese-English in the news domain. We used a large training set consisting of over 1 million sentences from various newswire corpora. This corpus is roughly the same as the one used for large-scale experiments by Chiang et al. (2005). To generate alignments, we used GIZA++ (Och and Ney, 2003). We symmetrized bidirectional alignments using the grow-diag-final heuristic (Koehn et al., 2003).

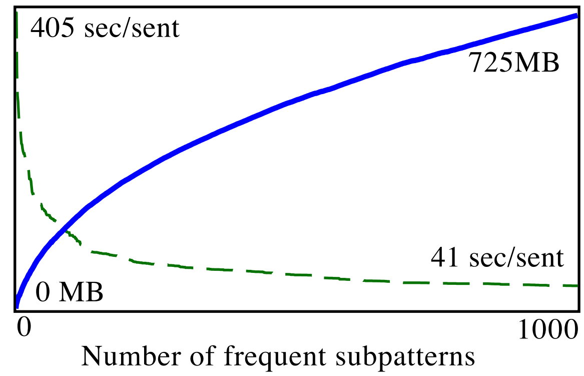

Figure 3: Effect of precomputation on memory use and processing time. Here we show only the memory requirements of the precomputed collocations.

We used the first 50 sentences of the NIST 2003 test set to compute timing results. All of our algorithms were implemented in Python 2.4. [10] Timing results are reported for machines with 8GB of memory and 4 3GHz Xeon processors running Red Hat linux 2.6.9. In order to understand the contributions of various improvements, we also ran the system with with various ablations. In the default setting, the prefix tree is constructed for each sentence to guide phrase lookup, and then discarded. To show the effect of caching we also ran the algorithm without discarding the prefix tree between sentences, resulting in full inter-sentence caching. The results are shown in Table 1. [11]

It is clear from the results that each of the optimizations is needed to sufficiently reduce lookup time to practical levels. Although this is still relatively slow, it is much closer to the decoding time of 10 seconds per sentence than the baseline.

[10] Python is an interpreted language and our implementations do not use any optimization features. It is therefore reasonable to think that a more efficient reimplementation would result in across-the-board speedups.

[11] The results shown here do not include the startup time required to load the data structures into memory. In our Python implementation this takes several minutes, which in principle should be amortized over the cost for each sentence. However, just as Zens and Ney (2007) do for phrase tables, we could compile our data structures into binary memory-mapped files, which can be read into memory in a matter of seconds. We are currently investigating this option in a C reimplementation.

Algorithms	Secs/Sent	Collocations
Baseline	2241.25	325548
Prefix Tree	1578.77	69994
Prefix Tree + precomputation	696.35	69994
Prefix Tree + double binary	405.02	69994
Prefix Tree + precomputation + double binary	40.77	69994
Prefix Tree with full caching + precomputation + double binary	30.70	67712

Table 1: Timing results and number of collocations computed for various combinations of algorithms. The runs using precomputation use the 1000 most frequent patterns.

7 Conclusions and Future Work

Our work solves a seemingly intractable problem and opens up a number of intriguing potential applications. Both Callison-Burch et al. (2005) and Zhang and Vogel (2005) use suffix arrays to relax the length constraints on phrase-based models. Our work enables this in hierarchical phrase-based models. However, we are interested in additional applications.

Recent work in discriminative learning for many natural language tasks, such as part-of-speech tagging and information extraction, has shown that feature engineering plays a critical role in these approaches. However, in machine translation most features can still be traced back to the IBM Models of 15 years ago (Lopez, 2007b). Recently, Lopez and Resnik (2006) showed that most of the features used in standard phrase-based models do not help very much. Our algorithms enable us to look up phrase pairs *in context*, which will allow us to compute interesting contextual features that can be used in discriminative learning algorithms to improve translation accuracy. Essentially, we can use the training data itself as an indirect representation of whatever features we might want to compute. This is not possible with table-based architectures.

Most of the data structures and algorithms discussed in this paper are widely used in bioinformatics, including suffix arrays, prefix trees, and suffix links (Gusfield, 1997). As discussed in §4.1, our problem is a variant of the approximate pattern matching problem. A major application of approximate pattern matching in bioinformatics is query processing in protein databases for purposes of sequencing, phylogeny, and motif identification.

Current MT models, including hierarchical models, translate by breaking the input sentence into small pieces and translating them largely independently. Using approximate pattern matching algorithms, we imagine that machine translation could be treated very much like search in a protein database. In this scenario, the goal is to select training sentences that match the input sentence as closely as possible, under some evaluation function that accounts for both matching and mismatched sequences, as well as possibly other data features. Once we have found the closest sentences we can translate the matched portions in their entirety, replacing mismatches with appropriate word, phrase, or hierarchical phrase translations as needed. This model would bring statistical machine translation closer to convergence with so-called *example-based translation*, following current trends (Marcu, 2001; Och, 2002). We intend to explore these ideas in future work.

Acknowledgements

I would like to thank Philip Resnik for encouragement, thoughtful discussions and wise counsel; David Chiang for providing the source code for his translation system; and Nitin Madnani, Smaranda Muresan and the anonymous reviewers for very helpful comments on earlier drafts of this paper. Any errors are my own. This research was supported in part by ONR MURI Contract FCPO.810548265 and the GALE program of the Defense Advanced Research Projects Agency, Contract No. HR0011-06-2-001. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the view of DARPA.

References

Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. 2004. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, Mar.

Ricardo Baeza-Yates and Alejandro Salinger. 2005. Experimental analysis of a fast intersection algorithm for sorted sequences. In M. Consens and G. Navarro, editors, *Proc. of SPIRE*, number 3772 in LNCS, pages 13–24, Berlin. Springer-Verlag.

Ricardo Baeza-Yates. 2004. A fast intersection algorithm for sorted sequences. In *Proc. of Combinatorial Pattern Matching*, number 3109 in LNCS, pages 400–408, Berlin. Springer-Verlag.

Chris Callison-Burch, Colin Bannard, and Josh Shroeder. 2005. Scaling phrase-based statistical machine translation to larger corpora and longer phrases. In *Proc. of ACL*, pages 255–262, Jun.

David Chiang, Adam Lopez, Nitin Madnani, Christof Monz, Philip Resnik, and Michael Subotin. 2005. The Hiero machine translation system: Extensions, evaluation, and analysis. In *Proc. of HLT-EMLP*, pages 779–786, Oct.

David Chiang. 2007. Hierarchical phrase-based translation. *Computational Linguistics*, 33(2). In press.

Marcello Federico and Nicola Bertoldi. 2006. How many bits are needed to store probabilities for phrase-based translation? In *Proc. of NAACL Workshop on Statistical Machine Translation*, pages 94–101, Jun.

Dan Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press.

Dan Klein and Christopher D. Manning. 2001. Parsing with treebank grammars: Empirical bounds, theoretical models, and the structure of the Penn Treebank. In *Proc. of ACL-EACL*, pages 330–337, Jul.

Philipp Koehn, Franz Josef Och, and Daniel Marcu. 2003. Statistical phrase-based translation. In *Proc. of HLT-NAACL*, pages 127–133, May.

Adam Lopez and Philip Resnik. 2006. Word-based alignment, phrase-based translation: What's the link? In *Proc. of AMTA*, pages 90–99, Aug.

Adam Lopez. 2007a. Hierarchical phrase-based translation with suffix arrays. Technical Report 2007-26, University of Maryland Institute for Advanced Computer Studies, May.

Adam Lopez. 2007b. A survey of statistical machine translation. Technical Report 2006-47, University of Maryland Institute for Advanced Computer Studies, Apr.

Udi Manber and Gene Myers. 1993. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948.

Daniel Marcu. 2001. Towards a unified approach to memory- and statistical-based machine translation. In *Proc. of ACL-EACL*, pages 378–385, Jul.

Gonzalo Navarro. 2001. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, Mar.

Franz Josef Och and Hermann Ney. 2003. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51, Mar.

Franz Josef Och and Hermann Ney. 2004. The alignment template approach to machine translation. *Computational Linguistics*, 30(4):417–449, Jun.

Franz Josef Och. 2002. *Statistical Machine Translation: From Single-Word Models to Alignment Templates*. Ph.D. thesis, RWTH Aachen, Oct.

Franz Josef Och. 2005. Statistical machine translation: The fabulous present and future. In *Proc. of ACL Workshop on Building and Using Parallel Texts*, Jun. Invited talk.

Chris Quirk, Arul Menezes, and Colin Cherry. 2005. Dependency treelet translation: Syntactically informed phrasal SMT. In *Proc. of ACL*, pages 271–279, Jun.

Mohammad Sohel Rahman, Costas S. Iliopoulos, Inbok Lee, Manal Mohamed, and William F. Smyth. 2006. Finding patterns with variable length gaps or don't cares. In *Proc. of COCOON*, Aug.

Michel Simard, Nicola Cancedda, Bruno Cavestro, Marc Dymetman, Eric Gaussier, Cyril Goutte, Kenji Yamada, Philippe Langlais, and Arne Mauser. 2005. Translating with non-contiguous phrases. In *Proc. of HLT-EMNLP*, pages 755–762, Oct.

Peter van Emde Boas, R. Kaas, and E. Zijlstra. 1977. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(2):99–127.

Richard Zens and Hermann Ney. 2007. Efficient phrase-table representation for machine translation with applications to online MT and speech translation. In *Proc. of HLT-NAACL*. To appear.

Ying Zhang and Stephan Vogel. 2005. An efficient phrase-to-phrase alignment model for arbitrarily long phrase and large corpora. In *Proc. of EAMT*, May.