Factorization of Synchronous Context-Free Grammars in Linear Time

Hao Zhang and **Daniel Gildea**
Computer Science Department
University of Rochester
Rochester, NY 14627

Abstract

Factoring a Synchronous Context-Free Grammar into an equivalent grammar with a smaller number of nonterminals in each rule enables synchronous parsing algorithms of lower complexity. The problem can be formalized as searching for the tree-decomposition of a given permutation with the minimal branching factor. In this paper, by modifying the algorithm of Uno and Yagiura (2000) for the closely related problem of finding all common intervals of two permutations, we achieve a linear time algorithm for the permutation factorization problem. We also use the algorithm to analyze the maximum SCFG rule length needed to cover hand-aligned data from various language pairs.

1 Introduction

A number of recent syntax-based approaches to statistical machine translation make use of Synchronous Context Free Grammar (SCFG) as the underlying model of translational equivalence. Wu (1997)'s Inversion Transduction Grammar, as well as tree-transformation models of translation such as Yamada and Knight (2001), Galley et al. (2004), and Chiang (2005) all fall into this category.

A crucial question for efficient computation in approaches based on SCFG is the length of the grammar rules. Grammars with longer rules can represent a larger set of reorderings between languages (Aho and Ullman, 1972), but also require greater computational complexity for word alignment algorithms based on synchronous parsing (Satta and Peserico, 2005). Grammar rules extracted from large parallel corpora by systems such as Galley et al. (2004) can be quite large, and Wellington et al. (2006) argue that complex rules are necessary by analyzing the coverage of gold-standard word alignments from different language pairs by various grammars.

However, parsing complexity depends not only on rule length, but also on the specific permutations represented by the individual rules. It may be possible to factor an SCFG with maximum rule length n into a simpler grammar with a maximum of k nonterminals in any one rule, if not all $n!$ permutations appear in the rules. Zhang et al. (2006) discuss methods for binarizing SCFGs, ignoring the non-binarizable grammars; in Section 2 we discuss the generalized problem of factoring to k-ary grammars for any k and formalize the problem as permutation factorization in Section 3.

In Section 4, we describe an $O(k \cdot n)$ left-to-right shift-reduce algorithm for analyzing permutations that can be k-arized. Its time complexity becomes $O(n^2)$ when k is not specified beforehand and the minimal k is to be discovered. Instead of linearly shifting in one number at a time, Gildea et al. (2006) employ a balanced binary tree as the control structure, producing an algorithm similar in spirit to merge-sort with a reduced time complexity of $O(n \log n)$. However, both algorithms rely on reduction tests on emerging spans which involve redundancies with the spans that have already been tested.

Uno and Yagiura (2000) describe a clever algorithm for the problem of finding all common intervals of two permutations in time $O(n + K)$, where K is the number of common intervals, which can itself be $\Omega(n^2)$. In Section 5, we adapt their approach to the problem of factoring SCFGs, and show that, given this problem definition, running time can be improved to $O(n)$, the optimum given the time needed to read the input permutation.

The methodology in Wellington et al. (2006) measures the complexity of word alignment using the number of gaps that are necessary for their synchronous parser which allows discontinuous spans to succeed in parsing. In Section 6, we provide a more direct measurement using the minimal branching factor yielded by the permutation factorization algorithm.

2 Synchronous CFG and Synchronous Parsing

We begin by describing the synchronous CFG formalism, which is more rigorously defined by Aho and Ullman (1972) and Satta and Peserico (2005).

We adopt the SCFG notation of Satta and Peserico (2005). Superscript *indices* in the right-hand side of grammar rules:

$$X \rightarrow X_1^{(1)} ... X_n^{(n)},\ X_{\pi(1)}^{(\pi(1))} ... X_{\pi(n)}^{(\pi(n))}$$

indicate that the nonterminals with the same index are linked across the two languages, and will eventually be rewritten by the same rule application. Each X_i is a variable which can take the value of any nonterminal in the grammar.

We say an SCFG is n-ary if and only if the maximum number of co-indexed nonterminals, i.e. the longest permutation contained in the set of rules, is of size n.

Given a synchronous CFG and a pair of input strings, we can apply a generalized CYK-style bottom up chart parser to build synchronous parse trees over the string pair. Wu (1997) demonstrates the case of binary SCFG parsing, where six string boundary variables, three for each language as in monolingual CFG parsing, interact with each other, yielding an $O(N^6)$ dynamic programming algorithm, where N is the string length, assuming the two paired strings are comparable in length. For an n-ary SCFG, the parsing complexity can be as high as $O(N^{n+4})$. The reason is even if we binarize on one side to maintain 3 indices, for many unfriendly permutations, at most $n + 1$ boundary variables in the other language are necessary.

The fact that this bound is exponential in the rule length n suggests that it is advantageous to reduce the length of grammar rules as much as possible. This paper focuses on converting an SCFG to the equivalent grammar with smallest possible maximum rule size. The algorithm processes each rule in the input grammar independently, and determines whether the rule can be factored into smaller SCFG rules by analyzing the rule's permutation π.

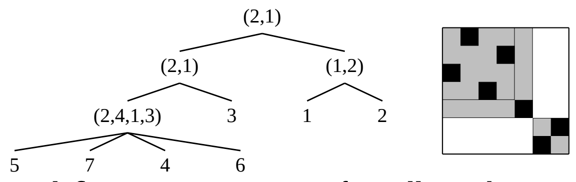
As an example, given the input rule:

$$[\, X \rightarrow A^{(1)} B^{(2)} C^{(3)} D^{(4)} E^{(5)} F^{(6)} G^{(7)},\quad X \rightarrow E^{(5)} G^{(7)} D^{(4)} F^{(6)} C^{(3)} A^{(1)} B^{(2)} \,] \qquad (1)$$

we consider the associated permutation:

$$(5, 7, 4, 6, 3, 1, 2)$$

We determine that this permutation can be factored into the following **permutation tree**:



We define permutation trees formally in the next section, but note here that nodes in the tree correspond to subsets of nonterminals that form a single continuous span in both languages, as shown by the shaded regions in the permutation matrix above. This tree can be converted into a set of output rules that are generatively equivalent to the original rule:

$$[\, X \rightarrow X_1^{(1)} X_2^{(2)},\ X \rightarrow X_2^{(2)} X_1^{(1)} \,]$$
$$[\, X_1 \rightarrow A^{(1)} B^{(2)},\ X_1 \rightarrow A^{(1)} B^{(2)} \,]$$
$$[\, X_2 \rightarrow C^{(1)} X_3^{(2)},\ X_2 \rightarrow X_3^{(2)} C^{(1)} \,]$$
$$[\, X_3 \rightarrow D^{(1)} E^{(2)} F^{(3)} G^{(4)},\quad X_3 \rightarrow E^{(2)} G^{(4)} D^{(1)} F^{(3)} \,]$$

where X_1, X_2 and X_3 are new nonterminals used to represent the intermediate states in which the synchronous nodes are combined. The factorized grammar is only larger than the original grammar by a constant factor.

3 Permutation Trees

We define the notion of permutation structure in this section. We define a **permuted sequence** as a permutation of n ($n \geq 1$) consecutive natural numbers.

A permuted sequence is said to be k**-ary parsable** if either of the following conditions holds:

1. The permuted sequence only has one number.

2. It has more than one number and can be segmented into k' ($k \geq k' \geq 2$) permuted sequences each of which is k-ary parsable, and the k' subsequences are arranged in an order identified by one of the $k'!$ permutations of k'.

This is a recursive definition, and we call the corresponding recursive structure over the entire sequence a k**-ary permutation tree**.

Our goal is to find out the k-ary permutation tree for a given permutation, where k is minimized.

4 Shift-reduce on Permutations

In this section, we present an $O(n \cdot k)$ algorithm which can be viewed as a need-to-be-optimized version of the linear time algorithm to be presented in the next section.

The algorithm is based on a shift-reduce parser, which maintains a stack for subsequences that have been discovered so far and loops over shift and reduce steps:

1. Shift the next number in the input permutation onto the stack.

2. Go down the stack from the top to the bottom. Whenever the top m subsequences satisfy the **partition property**, which says the total length of the m ($k \geq m \geq 2$) subsequences minus 1 is equal to the difference between the smallest number and the largest number contained in the m segments, make a reduction by gluing the m segments into one subsequence and restart reducing from the top of the new stack. Stop when no reduction is possible.

3. If there are remaining numbers in the input permutation, go to 1.

When we exit from the loop, if the height of the stack is 1, the input permutation of n has been reduced to a linear sequence of 1 to n, and parsing is successful. Otherwise, the input permutation of n cannot be parsed into a k-ary permutation tree.

Stack	Input	Operation
	5, 7, 4, 6, 3, 1, 2	shift
5	7, 4, 6, 3, 1, 2	shift
5, 7	4, 6, 3, 1, 2	shift
5, 7, 4	6, 3, 1, 2	shift
5, 7, 4, 6	3, 1, 2	reduce by (2,4,1,3)
[4...7]	3, 1, 2	shift
[4...7], 3	1, 2	reduce by (2,1)
[3...7]	1, 2	shift
[3...7], 1	2	shift
[3...7], 1, 2		reduce by (1,2)
[3...7], [1...2]		reduce by (2,1)
[1...7]		

Table 1: The execution trace of the shift-reduce parser on the input permutation $5, 7, 4, 6, 3, 1, 2$.

An example execution trace of the algorithm is shown in Table 1.

The partition property is a sufficient and necessary condition for the top m subsequences to be reducible. In order to check if the property holds, we need to compute the sum of the lengths of subsequences under consideration and the difference between the largest and smallest number in the covered region. We can incrementally compute both along with each step going down the stack. If m is bounded by k, we need $O(k)$ operations for each item shifted onto the stack. So, the algorithm runs in $O(n \cdot k)$.

We might also wish to compute the minimum k for which k-arization can be successful on an input permutation of n. We can simply keep doing reduction tests for every possible top region of the stack while going deeper in the stack to find the minimal reduction. In the worst case, each time we go down to the bottom of the increasingly higher stack without a successful reduction. Thus, in $O(n^2)$, we can find the minimum k-arization.

5 Linear Time Factorization

In this section, we show a linear time algorithm which shares the left-to-right and bottom-up control structure but uses more book-keeping operations to reduce unnecessary reduction attempts. The reason that our previous algorithm is asymptotically $O(n^2)$

is that whenever a new number is shifted in, we have to try out every possible new span ending at the new number. Do we need to try every possible span? Let us start with a motivating example. The permuted sequence $(5, 7, 4, 6)$ in Table 1 can only be reduced as a whole block. However, in the last algorithm, when 4 is shifted in, we make an unsuccessful attempt for the span on $(7, 4)$, knowing we are missing 5, which will not appear when we expand the span no matter how much further to the right. Yet we repeat the same mistake to try on 7 when 6 is scanned in by attempting on $(7, 4, 6)$. Such wasteful checks result in the quadratic behavior of the algorithm. The way the following algorithm differs from and outperforms the previous algorithm is exactly that it crosses out impossible candidates for reductions such as 7 in the example as early as possible.

Now we state our problem mathematically. We define a function whose value indicates the reducibility of each pair of positions (x, y) ($1 \leq x \leq y \leq n$):

$$f(x, y) = u(x, y) - l(x, y) - (y - x)$$

where

$$l(x, y) = \min_{i \in [x,y]} \pi(i)$$
$$u(x, y) = \max_{i \in [x,y]} \pi(i)$$

l records the minimum of the numbers that are permuted to from the positions in the region $[x, y]$. u records the maximum. Figure 1 provides the visualization of u, l, and f for the example permutation $(5, 7, 4, 6, 3, 1, 2)$. u and l can be visualized as stairs. u goes up from the right end to the left. l goes down. f is non-negative, but not monotonic in general. We can make a reduction on (x, y) if and only if $f(x, y) = 0$. This is the mathematical statement of the partition property in step 2 of the shift-reduce algorithm. u and l can be computed incrementally from smaller spans to larger spans to guarantee $O(1)$ operations for computing f on each new span of $[x, y]$ as long as we go bottom up. In the new algorithm, we will reduce the size of the search space of candidate position pairs (x, y) to be linear in n so that the whole algorithm is $O(n)$.

The algorithm has two main ideas:

- We filter x's to maintain the invariant that $f(x, y)$ ($x \leq y$) is monotonically decreasing with respect to x, over iterations on y (from 1 to n), so that any remaining values of x corresponding to valid reductions are clustered at the point where f tails off to zero. To put it another way, we never have to test invalid reductions, because the valid reductions have been sorted together for us.

- We make greedy reductions as in the shift-reduce algorithm.

In the new algorithm, we use a doubly linked list, instead of a stack, as the data structure that stores the candidate x's to allow for more flexible maintaining operations. The steps of the algorithm are as follows:

1. Increase the left-to-right index y by one and append it to the right end of the list.
2. Find the *pivot* x^* in the list which is minimum (leftmost) among x satisfying either $u(x, y - 1) < u(x, y)$ (exclusively) or $l(x, y - 1) > l(x, y)$.
3. Remove those x's that yield even smaller $u(x, y - 1)$ than $u(x^*, y - 1)$ or even larger $l(x, y - 1)$ than $l(x^*, y - 1)$. Those x's must be on the right of x^* if they exist. They must form a sub-list extending to the right end of the original x list.
4. Denote the x which is immediately to the left of x^* as x'. Repeatedly remove all x's such that $f(x, y) > f(x', y)$ where x is at the left end of the sub-list of x's starting from x^* extending to the right.
5. Go down the pruned list from the right end, output (x, y) until $f(x, y) > 0$. Remove x's such that $f(x, y) = 0$, sparing the smallest x which is the leftmost among all such x's on the list.
6. If there are remaining numbers in the input permutation, go to 1.

The tricks lie in step 3 and step 4, where bad candidate x's are filtered out. We use the following diagram to help readers understand the parts of x-list that the two steps are filtering on.

$$x_1, ..., x', \overbrace{x^*, ..., x_i}^{\textit{step 4}}, ..., \underbrace{x_j, ..., x_k}_{\textit{step 3}}, y$$

The steps from 2 to 4 are the operations that maintain the monotonic invariant which makes the reductions in step 5 as trivial as performing output. The stack-based shift-reduce algorithm has the same top-level structure, but lacks steps 2 to 4 so that in step 5 we have to winnow the entire list. Both algorithms scan left to right and examine potential reduction spans by extending the left endpoint from right to left given a right endpoint.

5.1 Example Execution Trace

An example of the algorithm's execution is shown in Figure 1. The evolution of $u(x, y)$, $l(x, y)$, and $f(x, y)$ is displayed for increasing y's (from 2 to 7). To identify reducible spans, we can check the plot of $f(x, y)$ to locate the (x, y) pairs that yield zero. The pivots found by step 2 of the algorithm are marked with $*$'s on the x-axis in the plot for u and l. The x's that are filtered out by step 3 or 4 are marked with horizontal bars across. We want to point out the interesting steps. When $y = 3$, $x^* = 1$, $x = 2$ needs to be crossed out by step 3 in the algorithm. When $y = 4$, $x^* = 3$, $x = 3$ itself is to be deleted by step 4 in the algorithm. $x = 4$ is removed at step 5 because it is the right end in the first reduction. On the other hand, $x = 4$ is also a bad starting point for future reductions. Notice that we also remove $x = 5$ at step 6, which can be a good starting point for reductions. But we exclude it from further considerations, because we want left-most reductions.

5.2 Correctness

Now we explain why the algorithm works. Both algorithms are greedy in the sense that at each scan point we exhaustively reduce all candidate spans to the leftmost possible point. It can be shown that greediness is safe for parsing permutations.

What we need to show is how the monotonic invariant holds and is valid. Now we sketch the proof. We want to show for all x_i remaining on the list, $f(x_i, y) \geq f(x_{i+1}, y)$. When $y = 1$, it is trivially true. Now we do the induction on y step by case analysis:

Case 1: If $x_i < x_{i+1} < x^*$, then $f(x_i, y) - f(x_i, y - 1) = -1$. The reason is if x_i is on the left of x^*, both $u(x_i, y)$ and $l(x_i, y)$ are not changed from the $y - 1$-th step, so the only difference is that $y - x_i$ has increased by one. Graphically, the f curve extending to the left of x^* shifts down a unit of 1. So, the monotonic property still holds to the left of x^*.

Case 2: If $x^* \leq x_i < x_{i+1}$, then $f(x_i, y) - f(x_i, y - 1) = c$ ($c \geq 0$). The reason is that after executing step 3 in the algorithm, the remaining x_i's have either their $u(x_i, y)$ shifted up uniformly with $l(x_i, y)$ being unchanged, or the symmetric case that $l(x_i, y)$ is shifted down uniformly without changing $u(x_i, y)$. In both cases, the difference between u and l increases by at least one unit to offset the one unit increase of $y - x_i$. The result is that the f curve extending from x^* to the right shifts up or remains the same.

Case 3: So the half curve of f on the left of x^* is shifting down and the half right curve on the right is shifting up, making it necessary to consider the case that x_i is on the left and x_{i+1} on the right. Fortunately, step 4 in the algorithm deals with this case explicitly by cutting down the head of the right half curve to smooth the whole curve into a monotonically decreasing one.

We still need one last piece for the proof, i.e., the validity of pruning. Is it possible we winnow off good x's that will become useful in later stages of y? The answer is no. The values we remove in step 3 and 4 are similar to the points indexing into the second and third numbers in the permuted sequence $(5, 7, 4, 6)$. Any span starting from these two points will not be reducible because the element 5 is missing.[1]

To summarize, we remove impossible left boundaries and keep good ones, resulting in the monotonicity of f function which in turn makes safe greedy reductions fast.

5.3 Implementation and Time Analysis

We use a doubly linked list to implement both the u and l functions, where list element includes a span of x values (shaded rectangles in Figure 1). Both lists can be doubly linked with the list of x's so that

[1]Uno and Yagiura (2000) prove the validity of step 3 and step 4 rigorously.

we can access the u function and l function at $O(1)$ time for each x. At the same time, if we search for x based on u or l, we can follow the stair functions, skipping many intermediate x's.

The total number of operations that occur at step 4 and step 5 is $O(n)$ since these steps just involve removing nodes on the x list, and only n nodes are created in total over the entire algorithm. To find x^*, we scan back from the right end of u list or l list. Due to step 3, each u (and l) element that we scan over is removed at this iteration. So the total number of operations accountable to step 2 and step 3 is bounded by the maximum number of nodes ever created on the u and l lists, which is also n.

5.4 Related Work

Our algorithm is based on an algorithm for finding all common intervals of two permutations (Uno and Yagiura, 2000). The difference[2] is in step 5, where we remove the embedded reducible x's and keep only the leftmost one; their algorithm will keep all of the reducible x's for future considerations so that in the example the number 3 will be able to involve in both the reduction $([4-7], 3)$ and $(3, [1-2])$. In the worst case, their algorithm will output a quadratic number of reducible spans, making the whole algorithm $O(n^2)$. Our algorithm is $O(n)$ in the worst case. We can also generate all common intervals by transforming the permutation tree output by our algorithm.

However, we are not the first to specialize the Uno and Yagiura algorithm to produce tree structures for permutations. Bui-Xuan et al. (2005) reached a linear time algorithm in the definition framework of PQ trees. PQ trees represent families of permutations that can be created by composing operations of scrambling subsequences according to any permutation (P nodes) and concatenating subsequences in order (Q nodes). Our definition of permutation tree can be thought of as a more specific version of a PQ tree, where the nodes are all labeled with a specific permutation which is not decomposable.

[2]The original Uno and Yagiura algorithm also has the minor difference that the scan point goes from right to left.

6 Experiments on Analyzing Word Alignments

We apply the factorization algorithm to analyzing word alignments in this section. Wellington et al. (2006) indicate the necessity of introducing discontinuous spans for synchronous parsing to match up with human-annotated word alignment data. The number of discontinuous spans reflects the structural complexity of the synchronous rules that are involved in building the synchronous trees for the given alignments. However, the more direct and detailed analysis would be on the branching factors of the synchronous trees for the aligned data.

Since human-aligned data has many-to-one word links, it is necessary to modify the alignments into one-to-one. Wellington et al. (2006) treat many-to-one word links disjunctively in their synchronous parser. We also commit to one of the many-one links by extracting a maximum match (Cormen et al., 1990) from the bipartite graph of the alignment. In other words, we abstract away the alternative links in the given alignment while capturing the backbone using the maximum number of word links.

We use the same alignment data for the five language pairs Chinese/English, Romanian/English, Hindi/English, Spanish/English, and French/English (Wellington et al., 2006). In Table 2, we report the number of sentences that are k-ary parsable but not $k-1$-ary parsable for increasing k's. Our analysis reveals that the permutations that are accountable for non-ITG alignments include higher order permutations such as $(3, 1, 5, 2, 4)$, albeit sparsely seen.

We also look at the number of terminals the non-binary synchronous nodes can cover. We are interested in doing so, because this can tell us how general these unfriendly rules are. Wellington et al. (2006) did a similar analysis on the English-English bitext. They found out the majority of non-ITG parsable cases are not local in the sense that phrases of length up to 10 are not helpful in covering the gaps. We analyzed the translation data for the five language pairs instead. Our result differs. The rightmost column in Table 2 shows that only a tiny percent of the non-ITG cases are significant in the sense that we can not deal with them through phrases or tree-flattening within windows of size 10.

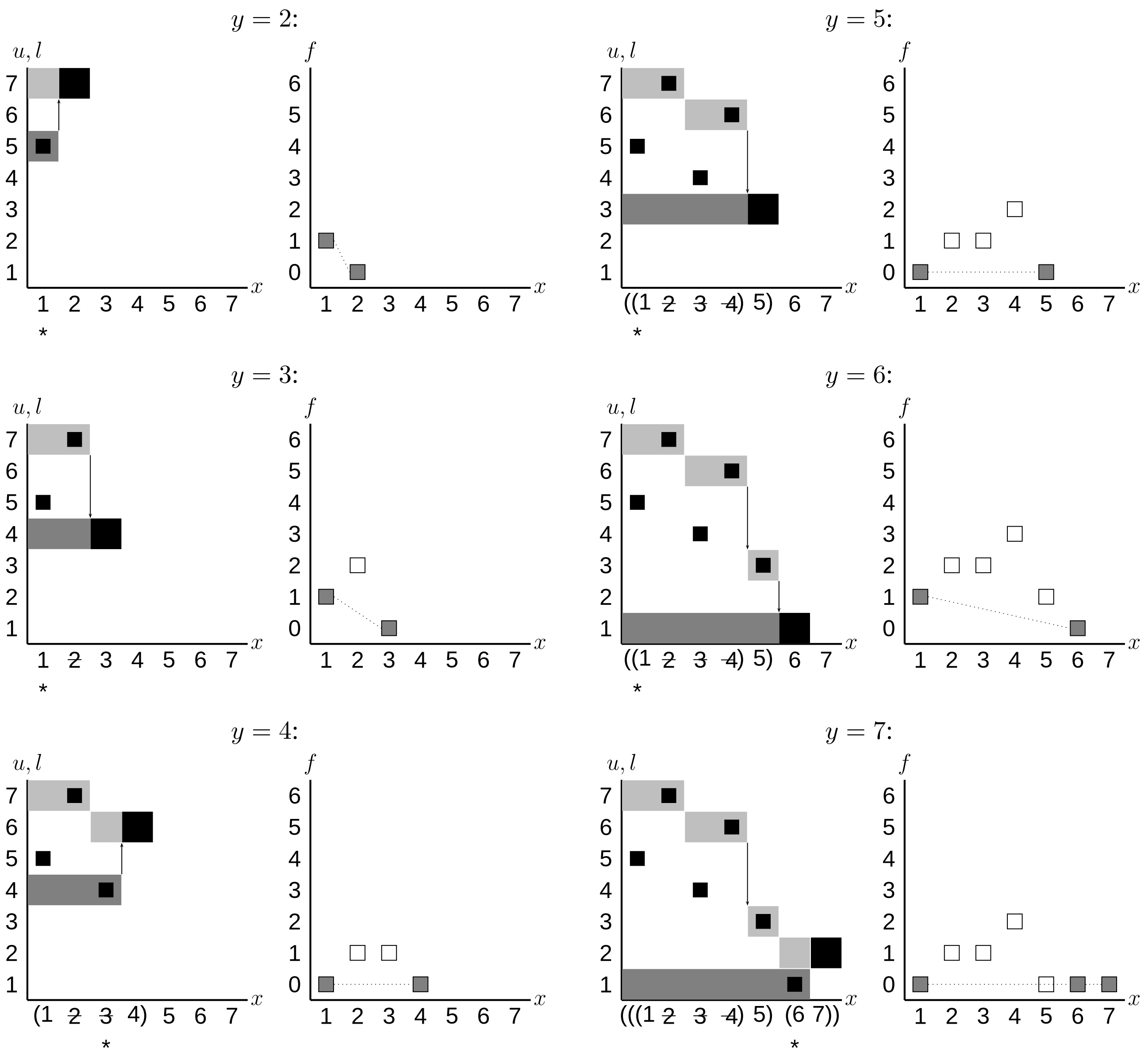


Figure 1: Evolution of $u(x, y)$, $l(x, y)$, and $f(x, y)$ as y goes from 2 to 7 for the permutation $(5, 7, 4, 6, 3, 1, 2)$. We use $*$ under the x-axis to indicate the x^*'s that are pivots in the algorithm. Useless x's are crossed out. x's that contribute to reductions are marked with either (on its left or) on its right. For the f function, we use solid boxes to plot the values of remaining x's on the list but also show the other f values for completeness.

	Branching Factor							
	1	2	4	5	6	7	10	≥ 4 (and covering > 10 words)
Chinese/English		451	30	4	5	1		7(1.4%)
Romanian/English		195	4					0
Hindi/English	3	85	1	1				0
Spanish/English		195	4					1(0.5%)
French/English		425	9	9	3		1	6(1.3%)

Table 2: Distribution of branching factors for synchronous trees on various language pairs.

7 Conclusion

We present a linear time algorithm for factorizing any n-ary SCFG rule into a set of k-ary rules where k is minimized. The algorithm speeds up an easy-to-understand shift-reduce algorithm, by avoiding unnecessary reduction attempts while maintaining the left-to-right bottom-up control structure. Empirically, we provide a complexity analysis of word alignments based on the concept of minimal branching factor.

References

Albert V. Aho and Jeffery D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling*, volume 1. Prentice-Hall, Englewood Cliffs, NJ.

Binh Minh Bui-Xuan, Michel Habib, and Christophe Paul. 2005. Revisiting T. Uno and M. Yagiura's algorithm. In *The 16th Annual International Symposium on Algorithms and Computation (ISAAC'05)*, pages 146–155.

David Chiang. 2005. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of the 43rd Annual Conference of the Association for Computational Linguistics (ACL-05)*, pages 263–270, Ann Arbor, Michigan.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to algorithms*. MIT Press, Cambridge, MA.

Michel Galley, Mark Hopkins, Kevin Knight, and Daniel Marcu. 2004. What's in a translation rule? In *Proceedings of the Human Language Technology Conference/North American Chapter of the Association for Computational Linguistics (HLT/NAACL)*.

Daniel Gildea, Giorgio Satta, and Hao Zhang. 2006. Factoring synchronous grammars by sorting. In *Proceedings of the International Conference on Computational Linguistics/Association for Computational Linguistics (COLING/ACL-06) Poster Session*, Sydney.

Giorgio Satta and Enoch Peserico. 2005. Some computational complexity results for synchronous context-free grammars. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP)*, pages 803–810, Vancouver, Canada, October.

Takeaki Uno and Mutsunori Yagiura. 2000. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309.

Benjamin Wellington, Sonjia Waxmonsky, and I. Dan Melamed. 2006. Empirical lower bounds on the complexity of translational equivalence. In *Proceedings of the International Conference on Computational Linguistics/Association for Computational Linguistics (COLING/ACL-06)*.

Dekai Wu. 1997. Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. *Computational Linguistics*, 23(3):377–403.

Kenji Yamada and Kevin Knight. 2001. A syntax-based statistical translation model. In *Proceedings of the 39th Annual Conference of the Association for Computational Linguistics (ACL-01)*, Toulouse, France.

Hao Zhang, Liang Huang, Daniel Gildea, and Kevin Knight. 2006. Synchronous binarization for machine translation. In *Proceedings of the Human Language Technology Conference/North American Chapter of the Association for Computational Linguistics (HLT/NAACL)*.