Factoring Synchronous Grammars By Sorting

Daniel Gildea
Computer Science Dept.
University of Rochester
Rochester, NY 14627

Giorgio Satta
Dept. of Information Eng'g
University of Padua
I-35131 Padua, Italy

Hao Zhang
Computer Science Dept.
University of Rochester
Rochester, NY 14627

Abstract

Synchronous Context-Free Grammars (SCFGs) have been successfully exploited as translation models in machine translation applications. When parsing with an SCFG, computational complexity grows exponentially with the length of the rules, in the worst case. In this paper we examine the problem of factorizing each rule of an input SCFG to a generatively equivalent set of rules, each having the smallest possible length. Our algorithm works in time $O(n \log n)$, for each rule of length n. This improves upon previous results and solves an open problem about recognizing permutations that can be factored.

1 Introduction

Synchronous Context-Free Grammars (SCFGs) are a generalization of the Context-Free Grammar (CFG) formalism to simultaneously produce strings in two languages. SCFGs have a wide range of applications, including machine translation, word and phrase alignments, and automatic dictionary construction. Variations of SCFGs go back to Aho and Ullman (1972)'s Syntax-Directed Translation Schemata, but also include the Inversion Transduction Grammars in Wu (1997), which restrict grammar rules to be binary, the synchronous grammars in Chiang (2005), which use only a single nonterminal symbol, and the Multitext Grammars in Melamed (2003), which allow independent rewriting, as well as other tree-based models such as Yamada and Knight (2001) and Galley et al. (2004).

When viewed as a rewriting system, an SCFG generates a set of string pairs, representing some translation relation. We are concerned here with the time complexity of parsing such a pair, according to the grammar. Assume then a pair with each string having a maximum length of N, and consider an SCFG G of size $|G|$, with a bound of n nonterminals in the right-hand side of each rule in a single dimension, which we call below the **rank** of G. As an upper bound, parsing can be carried out in time $O(|G|\, N^{n+4})$ by a dynamic programming algorithm maintaining continuous spans in one dimension. As a lower bound, parsing strategies with discontinuous spans in both dimensions can take time $\Omega(|G|\, N^{c\sqrt{n}})$ for unfriendly permutations (Satta and Peserico, 2005). A natural question to ask then is: What if we could reduce the rank of G, preserving the generated translation? As in the case of CFGs, one way of doing this would be to factorize each single rule into several rules of rank strictly smaller than n. It is not difficult to see that this would result in a new grammar of size at most $2 \cdot |G|$. In the time complexities reported above, we see that such a size increase would be more than compensated by the reduction in the degree of the polynomial in N. We thus conclude that a reduction in the rank of an SCFG would result in more efficient parsing algorithms, for most common parsing strategies.

In the general case, normal forms with bounded rank are not admitted by SCFGs, as shown in (Aho and Ullman, 1972). Nonetheless, an SCFG with a rank of n may not necessarily meet the worst case of Aho and Ullman (1972). It is then reasonable to ask if our SCFG G can be factorized, and what is the smallest rank $k < n$ that can be obtained in this way. This paper answers these two questions, by providing an algorithm that factorizes the rules of an input SCFG, resulting in a new, generatively equivalent, SCFG with rank k as low as possible. The algorithm works in time $O(n \log n)$ for each rule, regardless of the rank k of the factorized rules. As discussed above, in this way we achieve an improvement of the parsing time for SCFGs, obtaining an upper bound of $O(|G|\, N^{k+4})$ by using a parsing strategy that maintains continuous

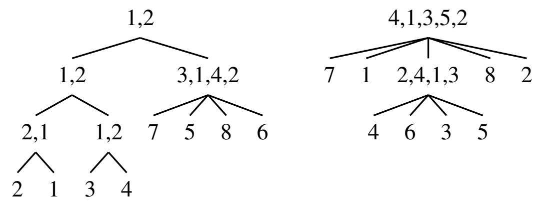


Figure 1: Two permutation trees. The permutations associated with the leaves can be produced by composing the permutations at the internal nodes.

spans in one dimension.

Previous work on this problem has been presented in Zhang et al. (2006), where a method is provided for casting an SCFG to a form with rank $k = 2$. If generalized to any value of k, that algorithm would run in time $O(n^2)$. We thus improve existing factorization methods by almost a factor of n. We also solve an open problem mentioned by Albert et al. (2003), who pose the question of whether irreducible, or *simple*, permutations can be recognized in time less than $\Theta(n^2)$.

2 Synchronous CFGs and permutation trees

We begin by describing the synchronous CFG formalism, which is more rigorously defined by Aho and Ullman (1972) and Satta and Peserico (2005). Let us consider strings defined over some set of nonterminal and terminal symbols, as defined for CFGs. We say that two such strings are **synchronous** if some bijective relation is given between the occurrences of the nonterminals in the two strings. A **synchronous context-free grammar** (SCFG) is defined as a CFG, with the difference that it uses synchronous rules of the form $[A_1 \to \alpha_1,\ A_2 \to \alpha_2]$, with A_1, A_2 nonterminals and α_1, α_2 synchronous strings. We can use production $[A_1 \to \alpha_1,\ A_2 \to \alpha_2]$ to rewrite any synchronous strings $[\gamma_{11}A_1\gamma_{12},\ \gamma_{21}A_2\gamma_{22}]$ into the synchronous strings $[\gamma_{11}\alpha_1\gamma_{12},\ \gamma_{21}\alpha_2\gamma_{22}]$, under the condition that the indicated occurrences of A_1 and A_2 be related by the bijection associated with the source synchronous strings. Furthermore, the bijective relation associated with the target synchronous strings is obtained by composing the relation associated with the source synchronous strings and the relation associated with synchronous pair $[\alpha_1, \alpha_2]$, in the most obvious way.

As in standard constructions that reduce the rank of a CFG, in this paper we focus on each single synchronous rule and factorize it into synchronous rules of lower rank. If we view the bijective relation associated with a synchronous rule as a permutation, we can further reduce our factorization problem to the problem of factorizing a permutation of arity n into the composition of several permutations of arity $k < n$. Such factorization can be represented as a tree of composed permutations, called in what follows a **permutation tree**. A permutation tree can be converted into a set of k-ary SCFG rules equivalent to the input rule. For example, the input rule:

$$\begin{aligned}[\,& X \to A^{(1)}B^{(2)}C^{(3)}D^{(4)}E^{(5)}F^{(6)}G^{(7)}H^{(8)},\\ & X \to B^{(2)}A^{(1)}C^{(3)}D^{(4)}G^{(7)}E^{(5)}H^{(8)}F^{(6)}\,]\end{aligned}$$

yields the permutation tree of Figure 1(left). Introducing a new grammar nonterminal X_i for each internal node of the tree yields an equivalent set of smaller rules:

$$\begin{aligned}&[\,X \to X_1^{(1)}X_2^{(2)},\ X \to X_1^{(1)}X_2^{(2)}\,]\\ &[\,X_1 \to X_3^{(1)}X_4^{(2)},\ X_1 \to X_3^{(1)}X_4^{(2)}\,]\\ &[\,X_3 \to A^{(1)}B^{(2)},\ X_3 \to B^{(2)}A^{(1)}\,]\\ &[\,X_4 \to C^{(1)}D^{(2)},\ X_4 \to C^{(1)}D^{(2)}\,]\\ &[\,X_2 \to E^{(1)}F^{(2)}G^{(3)}H^{(4)},\\ &\quad X_2 \to G^{(3)}E^{(1)}H^{(4)}F^{(2)}\,]\end{aligned}$$

In the case of stochastic grammars, the rule corresponding to the root of the permutation tree is assigned the original rule's probability, while all other rules, associated with new grammar nonterminals, are assigned probability 1. We process each rule of an input SCFG independently, producing an equivalent grammar with the smallest possible arity.

3 Factorization Algorithm

In this section we specify and discuss our factorization algorithm. The algorithm takes as input a permutation defined on the set $\{1, \cdots, n\}$, representing a rule of some SCFG, and provides a permutation tree of arity $k \leq n$ for that permutation, with k as small as possible.

Permutation trees covering a given input permutation are unambiguous with the exception of sequences of binary rules of the same type (either inverted or straight) (Albert et al., 2003). Thus, when factorizing a permutation into a permutation tree, it is safe to greedily reduce a subsequence into a new subtree as soon as a subsequence is found which represents a continuous span in both dimensions of the permutation matrix[1] associated with the input permutation. For space reasons, we omit the proof, but emphasize that any greedy reduction turns out to be either necessary, or equivalent to the other alternatives.

Any sequences of binary rules can be rearranged into a normalized form (e.g. always left-branching) as a postprocessing step, if desired.

The top-level structure of the algorithm exploits a divide-and-conquer approach, and is the same as that of the well-known mergesort algorithm (Cormen et al., 1990). We work on subsequences of the original permutation, and 'merge' neighboring subsequences into successively longer subsequences, combining two subsequences of length 2^i into a subsequence of length 2^{i+1} until we have built one subsequence spanning the entire permutation. If each combination of subsequences can be performed in linear time, then the entire permutation can be processed in time $O(n \log n)$. As in the case of mergesort, this is an application of the so-called master theorem (Cormen et al., 1990).

As the algorithm operates, we will maintain the invariant that we must have built all subtrees of the target permutation tree that are entirely within a given subsequence that has been processed. This is analogous to the invariant in mergesort that all processed subsequences are in sorted order. When we combine two subsequences, we need only build nodes in the tree that cover parts of both subsequences, but are entirely within the combined subsequence. Thus, we are looking for subtrees that span the midpoint of the combined subsequence, but have left and right boundaries within the boundaries of the combined subsequence. In what follows, this midpoint is called the **split point**.

From this invariant, we will be guaranteed to have a complete, correct permutation tree at the end of last subsequence combination. An example of the operation of the general algorithm is shown in Figure 2. The top-level structure of the algorithm is presented in function KARIZE of Figure 3.

There may be more than one reduction necessary spanning a given split point when combining two subsequences. Function MERGE in Figure 3 initializes certain data structures described below, and then checks for reductions repeatedly until no further reduction is possible. It looks first for the smallest reduction crossing the split point of the subsequences being combined. If SCAN, described below, finds a valid reduction, it is committed by calling REDUCE. If a reduction is found, we look for further reductions crossing either the left or right boundary of the new reduction, repeating until no further reductions are possible. Because we only need to find reductions spanning the original split point at a given combination step, this process is guaranteed to find all reductions needed.

[1]A permutation matrix is a way of representing a permutation, and is obtained by rearranging the row (or the columns) of an identity matrix, according to the permutation itself.

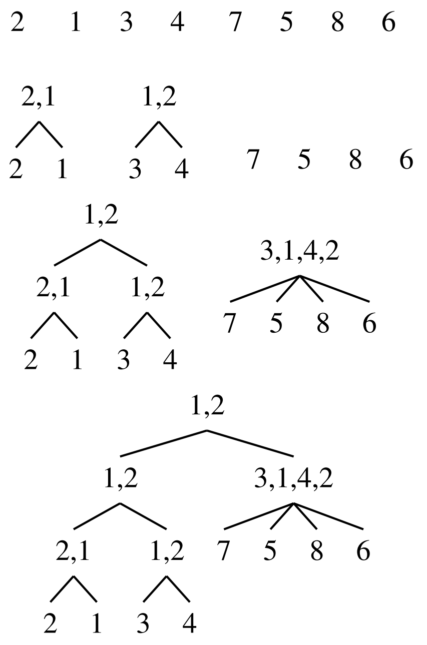


Figure 2: Recursive combination of permutation trees. Top row, the input permutation. Second row, after combination into sequences of length two, binary nodes have been built where possible. Third row, after combination into sequences of length four; bottom row, the entire output tree.

We now turn to the problem of identifying a specific reduction to be made across a split point, which involves identifying the reduction's left and right boundaries. Given a subsequence and candidate left and right boundaries for that subsequence, the validity of making a reduction over this span can be tested by verifying whether the span constitutes a **permuted sequence**, that is, a permutation of a contiguous sequence of integers. Since the starting permutation is defined on a set $\{1, 2, \cdots, n\}$, we have no repeated integers in our subsequences, and the above condi-

```
function KARIZE($\pi$)
                         ▷ initialize with identity mapping
    $h \leftarrow hmin \leftarrow hmax \leftarrow (0..|\pi|)$;
                                        ▷ mergesort core
    for $size \leftarrow 1$; $size \leq |\pi|$; $size \leftarrow size * 2$ do
        for $min \leftarrow 0$;
                $min < |\pi|-size+1$;
                $min \leftarrow min + 2 * size$ do
            $div = min + size - 1$;
            $max \leftarrow \min(|\pi|, min + 2*size - 1)$;
            MERGE($min$, $div$, $max$);
```

```
function MERGE($min$, $div$, $max$)
                                          ▷ initialize h
    sort $h[min..max]$ according to $\pi[i]$;
    sort $hmin[min..max]$ according to $\pi[i]$;
    sort $hmax[min..max]$ according to $\pi[i]$;
        ▷ merging sorted list takes linear time
                                          ▷ initialize v
    for $i \leftarrow min$; $i \leq max$; $i \leftarrow i + 1$ do
        $v[\,h[i]\,] \leftarrow i$;
          ▷ check if start of new reduced block
        if $i = min$ or
                    $hmin[i] \neq hmin[i-1]$ then
            $vmin \leftarrow i$;
        $vmin[\,h[i]\,] \leftarrow vmin$;
    for $i \leftarrow max$; $i \geq min$; $i \leftarrow i - 1$ do
          ▷ check if start of new reduced block
        if $i = max$ or
                    $hmax[i] \neq hmax[i+1]$ then
            $vmax \leftarrow i$ ;
        $vmax[\,h[i]\,] \leftarrow vmax$;
                                 ▷ look for reductions
    if SCAN($div$) then
        REDUCE(scanned reduction);
        while SCAN($left$) or SCAN($right$) do
            REDUCE(smaller reduction);
```

```
function REDUCE($left$, $right$, $bot$, $top$)
    for $i \leftarrow bot..top$ do
        $hmin[i] \leftarrow left$;
        $hmax[i] \leftarrow right$;
    for $i \leftarrow left..right$ do
        $vmin[i] \leftarrow bot$;
        $vmax[i] \leftarrow top$;
    print "reduce:" $left..right$ ;
```

Figure 3: KARIZE: Top level of algorithm, identical to that of mergesort. MERGE: combines two subsequences of size 2^i into new subsequence of size 2^{i+1}. REDUCE: commits reduction by updating *min* and *max* arrays.

tion can be tested by scanning the span in question, finding the minimum and maximum integers in the span, and checking whether their difference is equal to the length of the span minus one. Below we call this condition the **reduction test**. As an example of the reduction test, consider the subsequence $(7, 5, 8, 6)$, and take the last three elements, $(5, 8, 6)$, as a candidate span. We see that 5 and 8 are the minimum and maximum integers in the corresponding span, respectively. We then compute $8 - 5 = 3$, while the length of the span minus one is 2, implying that no reduction is possible. However, examining the entire subsequence, the minimum is 5 and the maximum is 8, and $8 - 5 = 3$, which is the length of the span minus one. We therefore conclude that we can reduce that span by means of some permutation, that is, parse the span by means of a node in the permutation tree. This reduction constitutes the 4-ary node in the permutation tree of Figure 2.

A trivial implementation of the reduction test would be to tests all combinations of left and right boundaries for the new reduction. Unfortunately, this would take time $\Omega(n^2)$ for a single subsequence combination step, whereas to achieve the overall $O(n \log n)$ complexity we need linear time for each combination step.

It turns out that the boundaries of the next reduction, covering a given split point, can be computed in linear time with the technique shown in function SCAN of Figure 5. We start with left and right candidate boundaries at the two points immediately to the left and right of the split point, and then repeatedly check whether the current left and right boundaries identify a permuted sequence by applying the reduction test, and move the left and right boundaries outward as necessary, as soon as 'missing' integers are identified outside the current boundaries, as explained below. We will show that, as we move outward, the number of possible configurations achieved for the positions of the left and the right boundaries is linearly bounded in the length of the combined subsequence (as opposed to quadratically bounded).

In order to efficiently implement the above idea, we will in fact maintain four boundaries for the candidate reduction, which can be visualized as the left, right, top and bottom boundaries in the permutation matrix. No explicit representation of the permutation matrix itself is constructed, as that would require quadratic time. Rather, we

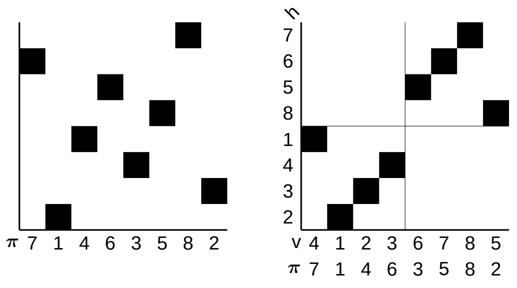


Figure 4: Permutation matrix for input permutation π (left) and within-subsequence permutation *v* (right) for subsequences of size four.

maintain two arrays: *h*, which maps from vertical to horizontal positions within the current subsequence, and *v* which maps from horizontal to vertical positions. These arrays represent the within-subsequence permutation obtained by sorting the elements of each subsequence according to the input permutation, while keeping each element within its block, as shown in Figure 4.

Within each subsequence, we alternate between scanning horizontally from left to right, possibly extending the top and bottom boundaries (Figure 5 lines 9 to 14), and scanning vertically from bottom to top, possibly extending the left and right boundaries (lines 20 to 26). Each extension is forced when, looking at the within-subsequence permutation, we find that some element is within the current boundaries in one dimension but outside the boundaries in the other. If the distance between vertical boundaries is larger in the input permutation than in the subsequence permutation, necessary elements are missing from the current subsequence and no reduction is possible at this step (line 18). When all necessary elements are present in the current subsequence and no further extensions are necessary to the boundaries (line 30), we have satisfied the reduction test on the input permutation, and make a reduction.

The trick used to keep the iterative scanning linear is that we *skip* the subsequence scanned on the previous iteration on each scan, in both the horizontal and vertical directions. Lines 13 and 25 of Figure 5 perform this skip by advancing the *x* and *y* counters past previously scanned regions. Considering the horizontal scan of lines 9 to 14, in a given iteration of the while loop, we scan only the items between *newleft* and *left* and between *right* and *newright*. On the next iteration of the while loop, the *newleft* boundary has moved further to the left,

```
1:  function SCAN ($div$)
2:      $left \leftarrow -\infty$;
3:      $right \leftarrow -\infty$;
4:      $newleft \leftarrow div$;
5:      $newright \leftarrow div + 1$ ;
6:      $newtop \leftarrow -\infty$;
7:      $newbot \leftarrow \infty$;
8:      while 1 do
                                        ▷ horizontal scan
9:          for $x \leftarrow newleft$; $x \leq newright$ ; do
10:             $newtop \leftarrow \max(newtop, vmax[x])$;
11:             $newbot \leftarrow \min(newbot, vmin[x])$;
                                ▷ skip to end of reduced block
12:             $x \leftarrow hmax[vmin[x]] + 1$;
                            ▷ skip section scanned on last iter
13:             if $x = left$ then
14:                 $x \leftarrow right + 1$;
15:         $right \leftarrow newright$;
16:         $left \leftarrow newleft$;
                                        ▷ the reduction test
17:         if $newtop - newbot <$
18:             $\pi[h[newtop]] - \pi[h[newbot]]$ then
19:             return (0);
                                        ▷ vertical scan
20:         for $y \leftarrow newbot$; $y \leq newtop$ ; do
21:             $newright \leftarrow$
22:                         $\max(newright, hmax[y])$;
23:             $newleft \leftarrow \min(newleft, hmin[y])$;
                                ▷ skip to end of reduced block
24:             $y \leftarrow vmax[hmin[y]] + 1$;
                            ▷ skip section scanned on last iter
25:             if $y = bot$ then
26:                 $y \leftarrow top + 1$;
27:         $top \leftarrow newtop$;
28:         $bot \leftarrow newbot$;
                        ▷ if no change to boundaries, reduce
29:         if $newright = right$
30:                         and $newleft = left$ then
31:             return (1, $left$, $right$, $bot$, $top$);
```

Figure 5: Linear time function to check for a single reduction at split point *div*.

while the variable *left* takes the previous value of *newleft*, ensuring that the items scanned on this iteration are distinct from those already processed. Similarly, on the right edge we scan new items, between *right* and *newright*. The same analysis applies to the vertical scan. Because each item in the permutation is scanned only once in the vertical direction and once in the horizontal direction, the entire call to SCAN takes linear time, regardless of the number of iterations of the while loop that are required.

We must further show that each call to MERGE takes only linear time, despite that fact that it may involve many calls to SCAN. We accomplish this by introducing a second type of skipping in the scans, which advances past any previously reduced block in a single step. In order to skip past previous reductions, we maintain (in function REDUCE) auxiliary arrays with the minimum and maximum positions of the largest block each point has been reduced to, in both the horizontal and vertical dimensions. We use these data structures (*hmin*, *hmax*, *vmin*, *vmax*) when advancing to the next position of the scan in lines 12 and 24 of Figure 5. Because each call to SCAN skips items scanned by previous calls, each item is scanned at most twice across an entire call to MERGE, once when scanning across a new reduction's left boundary and once when scanning across the right boundary, guaranteeing that MERGE completes in linear time.

4 An Example

In this section we examine the operation of the algorithm on a permutation of length eight, resulting in the permutation tree of Figure 1(right). We will build up our analysis of the permutation by starting with individual items of the input permutation and building up subsequences of length 2, 4, and finally 8. In our example permutation, $(7, 1, 4, 6, 3, 5, 8, 2)$, no reductions can be made until the final combination step, in which one permutation of size 4 is used, and one of size 5.

We begin with the input permutation along the bottom of Figure 6a. We represent the intermediate data structures *h*, *hmin*, and *hmax* along the vertical axis of the figure; these three arrays are all initialized to be the sequence $(1, 2, \cdots, 8)$.

Figure 6b shows the combination of individual items into subsequences of length two. Each new subsequence of the *h* array is sorted according to

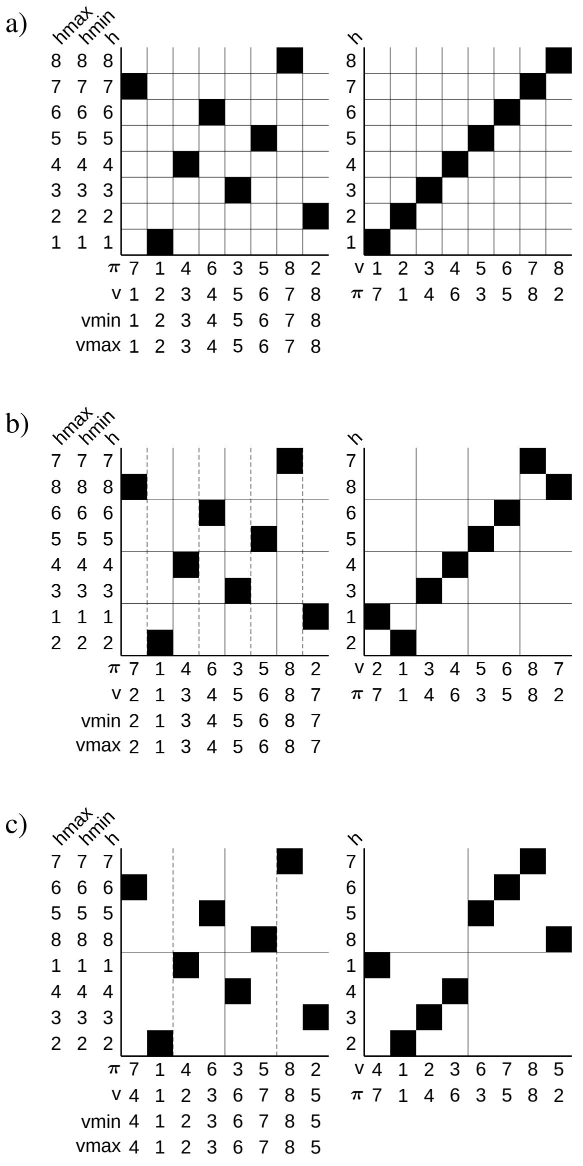
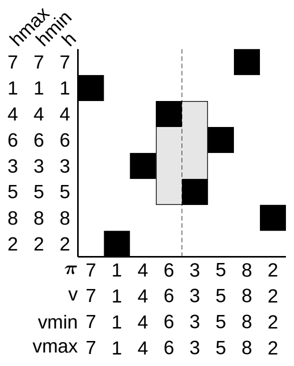


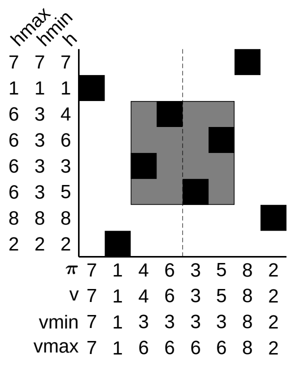
Figure 6: Steps in an example computation, with input permutation π on left and within-subsequence permutation described by *v* array on right. Panel (a) shows initial blocks of unit size, (b) shows combination of unit blocks into blocks of size two, and (c) size two into size four. No reductions are possible in these stages; example continued in next figure.

a)



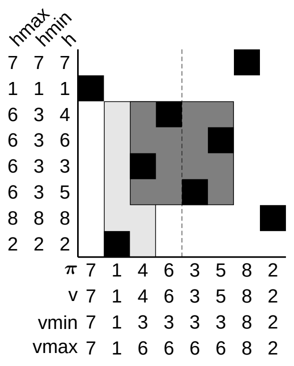
Left and right boundaries are initialized to be adjacent to horizontal split point.

b)



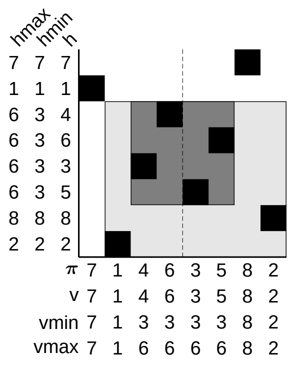
Vertical scan shows left and right boundaries must be extended. Permutation of size four is reduced.

c)



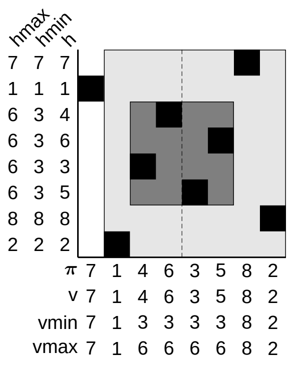
Search for next reduction: left and right boundaries initialized to be adjacent to left edge of previous reduction.

d)



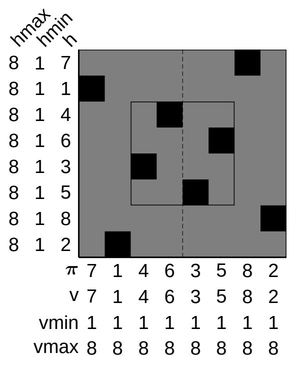
Vertical scan shows right boundary must be extended.

e)



Horizontal scan shows top boundary must be extended.

f)



Vertical scan shows left boundary must be extended. Permutation of size five is reduced.

Figure 7: Steps in scanning for final combination of subsequences, where $v = \pi$. Area within current left, right, top and bottom boundaries is shaded; darker shading indicates a reduction. In each scan, the span scanned in the previous panel is skipped over.

the vertical position of the dots in the corresponding columns. Thus, because $\pi[7] = 8 > \pi[8] = 2$, we swap 7 and 8 in the *h* array. The algorithm checks whether any reductions can be made at this step by computing the difference between the integers on each side of each split point. Because none of the pairs of integers in are consecutive, no reductions are made at this step.

Figure 6c shows the combination the pairs into subsequences of length four. The two split points to be examined are between the second and third position, and the sixth and seventh position. Again, no reductions are possible.

Finally we combine the two subsequences of length four to complete the analysis of the entire permutation. The split point is between the fourth and fifth positions of the input permutation, and in the first horizontal scan of these two positions, we see that $\pi[4] = 6$ and $\pi[5] = 3$, meaning our top boundary will be 6 and our bottom boundary 3, shown in Figure 7a. Scanning vertically from position 3 to 6, we see horizontal positions 5, 3, 6, and 4, giving the minimum, 3, as the new left boundary and the maximum, 6, as the new right boundary, shown in Figure 7b. We now perform another horizontal scan starting at position 3, but then jumping directly to position 6, as horizontal positions 4 and 5 were scanned previously. After this scan, the minimum vertical position seen remains 3, and the maximum vertical position is still 6. At this point, because we have the same boundaries as on the previous scan, we can stop and verify whether the region determined by our current boundaries has the same length in the vertical and horizontal dimensions. Both dimensions have length four, meaning that we have found a subsequence that is continuous in both dimensions and can safely be reduced, as shown in Figure 6d.

After making this reduction, we update the *hmin* array to have all 3's for the newly reduced span, and update *hmax* to have all sixes. We then check whether further reductions are possible covering this split point. We repeat the process of scanning horizontally and vertically in Figure 7c-f, this time skipping the span just reduced. One further reduction is possible, covering the entire input permutation, as shown in Figure 7f.

5 Conclusion

The algorithm above not only identifies whether a permutation can be factored into a composition of permutations, but also returns the factorization that minimizes the largest rule size, in time $O(n \log n)$. The factored SCFG with rules of size at most k can be used to synchronously parse in time $O(N^{k+4})$ by dynamic programming with continuous spans in one dimension.

As mentioned in the introduction, the optimal parsing strategy for SCFG rules with a given permutation may involve dynamic programming states with discontinuous spans in both dimensions. Whether these optimal parsing strategies can be found efficiently remains an interesting open problem.

Acknowledgments This work was partially supported by NSF ITR IIS-09325646 and NSF ITR IIS-0428020.

References

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

M. H. Albert, M. D. Atkinson, and M. Klazar. 2003. The enumeration of simple permutations. *Journal of Integer Sequences*, 6(03.4.4):18 pages.

David Chiang. 2005. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of ACL-05*, pages 263–270.

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 HLT/NAACL*.

I. Dan Melamed. 2003. Multitext grammars and synchronous parsers. In *Proceedings of HLT/NAACL*.

Giorgio Satta and Enoch Peserico. 2005. Some computational complexity results for synchronous context-free grammars. In *Proceedings of HLT/EMNLP*, pages 803–810.

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 ACL-01*.

Hao Zhang, Liang Huang, Daniel Gildea, and Kevin Knight. 2006. Synchronous binarization for machine translation. In *Proceedings of HLT/NAACL*.