A CYK+ Variant for SCFG Decoding Without a Dot Chart

Rico Sennrich
School of Informatics
University of Edinburgh
10 Crichton Street
Edinburgh EH8 9AB
Scotland, UK
v1rsennr@staffmail.ed.ac.uk

Abstract

While CYK+ and Earley-style variants are popular algorithms for decoding unbinarized SCFGs, in particular for syntax-based Statistical Machine Translation, the algorithms rely on a so-called dot chart which suffers from a high memory consumption. We propose a recursive variant of the CYK+ algorithm that eliminates the dot chart, without incurring an increase in time complexity for SCFG decoding. In an evaluation on a string-to-tree SMT scenario, we empirically demonstrate substantial improvements in memory consumption and translation speed.

1 Introduction

SCFG decoding can be performed with monolingual parsing algorithms, and various SMT systems implement the CYK+ algorithm or a close Earley-style variant (Zhang et al., 2006; Koehn et al., 2007; Venugopal and Zollmann, 2009; Dyer et al., 2010; Vilar et al., 2012). The CYK+ algorithm (Chappelier and Rajman, 1998) generalizes the CYK algorithm to n-ary rules by performing a dynamic binarization of the grammar during parsing through a so-called dot chart. The construction of the dot chart is a major cause of space inefficiency in SCFG decoding with CYK+, and memory consumption makes the algorithm impractical for long sentences without artificial limits on the span of chart cells.

We demonstrate that, by changing the traversal through the main parse chart, we can eliminate the dot chart from the CYK+ algorithm at no computational cost for SCFG decoding. Our algorithm improves space complexity, and an empirical evaluation confirms substantial improvements in memory consumption over the standard CYK+ algorithm, along with remarkable gains in speed.

This paper is structured as follows. As motivation, we discuss some implementation needs and complexity characteristics of SCFG decoding We then describe our algorithm as a variant of CYK+, and finally perform an empirical evaluation of memory consumption and translation speed of several parsing algorithms.

2 SCFG Decoding

To motivate our algorithm, we want to highlight some important differences between (monolingual) CFG parsing and SCFG decoding.

Grammars in SMT are typically several orders of magnitude larger than for monolingual parsing, partially because of the large amounts of training data employed to learn SCFGs, partially because SMT systems benefit from using contextually rich rules rather than only minimal rules (Galley et al., 2006). Also, the same right-hand-side rule on the source side can be associated with many translations, and different (source and/or target) left-hand-side symbols. Consequently, a compact representation of the grammar is of paramount importance.

We follow the implementation in the Moses SMT toolkit (Koehn et al., 2007) which encodes an SCFG as a trie in which each node represents a (partial or completed) rule, and a node has outgoing edges for each possible continuation of the rule in the grammar, either a source-side terminal symbol or pair of non-terminal-symbols. If a node represents a completed rule, it is also associated with a collection of left-hand-side symbols and the associated target-side rules and probabilities. A trie data structure allows for an efficient grammar lookup, since all rules with the same pre-

fix are compactly represented by a single node.

Rules are matched to the input in a bottom-up-fashion as described in the next section. A single rule or rule prefix can match the input many times, either by matching different spans of the input, or by matching the same span, but with different subspans for its non-terminal symbols. Each production is uniquely identified by a span, a grammar trie node, and back-pointers to its subderivations. The same is true for a partial production (*dotted item*).

A key difference between monolingual parsing and SCFG decoding, whose implications on time complexity are discussed by Hopkins and Langmead (2010), is that SCFG decoders need to consider language model costs when searching for the best derivation of an input sentence. This critically affects the parser's ability to discard dotted items early. For CFG parsing, we only need to keep one partial production per rule prefix and span, or k for k-best parsing, selecting the one(s) whose subderivations have the lower cost in case of ambiguity. For SCFG decoding, the subderivation with the higher local cost may be the globally better choice after taking language model costs into account. Consequently, SCFG decoders need to consider multiple possible productions for the same rule and span.

Hopkins and Langmead (2010) provide a runtime analysis of SCFG decoding, showing that time complexity depends on the number of choice points in a rule, i.e. rule-initial, consecutive, or rule-final non-terminal symbols.[1] The number of choice points (or *scope*) gives an upper bound to the number of productions that exist for a rule and span. If we define the scope of a grammar G to be the maximal scope of all rules in the grammar, decoding can be performed in $O(n^{scope(G)})$ time. If we retain all partial productions of the same rule prefix, this also raises the space complexity of the dot chart from $O(n^2)$ to $O(n^{scope(G)})$. [2]

Crucially, the inclusion of language model costs both increases the space complexity of the dot chart, and removes one of its benefits, namely the ability to discard partial productions early without risking search errors. Still, there is a second way

[1]Assuming that there is a constant upper bound on the frequency of each symbol in the input sentence, and on the length of rules.

[2]In a left-to-right construction of productions, a rule prefix of a scope-x rule may actually have scope $x + 1$, namely if the rule prefix ends in a non-terminal, but the rule does not.

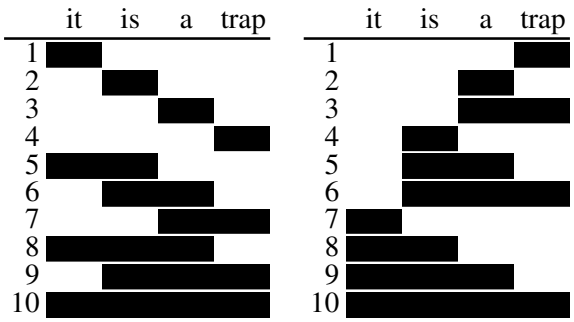

Figure 1: Traditional CYK/CYK+ chart traversal order (left) and proposed order (right).

in which a dot chart saves computational cost in the CYK+ algorithm. The exact chart traversal order is underspecified in CYK parsing, the only requirement being that all subspans of a given span need to be visited before the span itself. CYK+ or Earley-style parsers typically traverse the chart bottom-up left-to-right, as in Figure 1 (left). The same partial productions are visited throughout time during chart parsing, and storing them in a dot chart saves us the cost of recomputing them. For example, step 10 in Figure 1 (left) re-uses partial productions that were found in steps 1, 5 and 8.

We propose to specify the chart traversal order to be right-to-left, depth-first, as illustrated on the right-hand-side in Figure 1. This traversal order groups all cells with the same start position together, and offers a useful guarantee. For each span, all spans that start at a later position have been visited before. Thus, whenever we generate a partial production, we can immediately explore all of its continuations, and then discard the partial production. This eliminates the need for a dot chart, without incurring any computational cost. We could also say that the dot chart exists in a minimal form with at most one item at a time, and a space complexity of $O(1)$. We proceed with a description of the proposed algorithm, contrasted with the closely related CYK+ algorithm.

3 Algorithm

3.1 The CYK+ algorithm

We here summarize the CYK+ algorithm, originally described by Chappelier and Rajman (1998).[3]

[3]Chappelier and Rajman (1998) add the restriction that rules may not be partially lexicalized; our description of CYK+, and our own algorithm, do not place this restriction.

The main data structure during decoding is a chart with one cell for each span of words in an input string $w_1...w_n$ of length n. Each cell $T_{i,j}$ corresponding to the span from w_i to w_j contains two lists of items:[4]

- a list of type-1 items, which are non-terminals (representing productions).
- a list of type-2 items (dotted items), which are strings of symbols α that parse the substring $w_i...w_j$ and for which there is a rule in the grammar of the form $A \rightarrow \alpha\beta$, with β being a non-empty string of symbols. Such an item may be completed into a type-1 item at a future point, and is denoted $\alpha\bullet$.

For each cell (i, j) of the chart, we perform the following steps:

1. if $i = j$, search for all rules $A \rightarrow w_i\gamma$. If γ is empty, add A to the type-1 list of cell (i, j); otherwise, add $w_i\bullet$ to the type-2 list of cell (i, j).
2. if $j > i$, search for all combinations of a type-2 item $\alpha\bullet$ in a cell (i, k) and a type-1 item B in a cell $(k+1, j)$ for which a rule of the form $A \rightarrow \alpha B\gamma$ exists.[5] If γ is empty, add the rule to the type-1 list of cell (i, j); otherwise, add $\alpha B\bullet$ to the type-2 list of cell (i, j).
3. for each item B in the type-1 list of the cell (i, j), if there is a rule of the form $A \rightarrow B\gamma$, and γ is non-empty, add $B\bullet$ to the type-2 list of cell (i, j).

3.2 Our algorithm

The main idea behind our algorithm is that we can avoid the need to store type-2 lists if we process the individual cells in a right-to-left, depth-first order, as illustrated in Figure 1. Rules are still completed left-to-right, but processing the rightmost cells first allows us to immediately extend partial productions into full productions instead of storing them in memory.

We perform the following steps for each cell.

1. if $i = j$, if there is a rule $A \rightarrow w_i$, add A to the type-1 list of cell (i, j).
2. if $j > i$, search for all combinations of a type-2 item $\alpha\bullet$ and a type-1 item B in a cell (j, k), with $j \leq k \leq n$ for which a rule of the form $C \rightarrow \alpha B\gamma$ exists. In the initial call, we allow $\alpha\bullet = A\bullet$ for any type-1 item A in cell $(i, j-1)$.[6] If γ is empty, add C to the type-1 list of cell (i, k); otherwise, recursively repeat this step, using $\alpha B\bullet$ as $\alpha\bullet$ and $k + 1$ as j.

To illustrate the difference between the two algorithms, let us consider the chart cell $(1, 2)$, i.e. the chart cell spanning the substring *it is*, in Figure 1, and let us assume the following grammar:

S	$\rightarrow$	NP V NP	V	$\rightarrow$	*is*
NP	$\rightarrow$	ART NN	ART	$\rightarrow$	*a*
NP	$\rightarrow$	*it*	NN	$\rightarrow$	*trap*

In both algorithms, we can combine the symbols NP from cell $(1, 1)$ and V from cell $(2, 2)$ to partially parse the rule S $\rightarrow$ NP V NP. However, in CYK+, we cannot yet know if the rule can be completed with a cell $(3, x)$ containing symbol NP, since the cell $(3, 4)$ may be processed after cell $(1, 2)$. Thus, the partial production is stored in a type-2 list for later processing.

In our algorithm, we require all cells $(3, x)$ to be processed before cell $(1, 2)$, so we can immediately perform a recursion with α = NP V and $j = 3$. In this recursive step, we search for a symbol NP in any cell $(3, x)$, and upon finding it in cell $(3, 4)$, add S as type-1 item to cell $(1, 4)$.

We provide side-by-side pseudocode of the two algorithms in Figure 2.[7] The algorithms are aligned to highlight their similarity, the main difference between them being that type-2 items are added to the dot chart in CYK+, and recursively consumed in our variant. An attractive property of the dynamic binarization in CYK+ is that each partial production is constructed exactly once, and can be re-used to find parses for cells that cover a larger span. Our algorithm retains this property. Note that the chart traversal order is different between the algorithms, as illustrated earlier in Figure 1. While the original CYK+ algorithm works with either chart traversal order, our recursive vari-

However, our description excludes non-lexical unary rules, and epsilon rules.

[4] For simplicity, we describe a monolingual acceptor.

[5] To allow mixed-terminal rules, we also search for $B = w_j$ if $j = k + 1$.

[6] To allow mixed-terminal rules, we also allow $\alpha\bullet = w_i\bullet$ if $j = i + 1$, and $B = w_j$ if $k = j$.

[7] Some implementation details are left out for simplicity. For instance, note that terminal and non-terminal grammar trie edges can be kept separate to avoid iterating over all terminal edges.

Algorithm 1: CYK+

```
Input: array w of length N
initialize chart[N, N], collections[N, N],
dotchart[N]
root ← root node of grammar trie
for span in [1..N]:
  for i in [1..(N-span+1)]:
    j ← i+span-1
    if i = j:                                  #step 1
      if (w[i], X) in arc[root]:
        addToChart(X, i, j)
    else:
      for B in chart[i, j-1]:                  #step 3
        if (B, X) in arc[root]:
          if arc[X] is not empty:
            add (X, j-1) to dotchart[i]
      for (a, k) in dotchart[i]:               #step 2

        if k+1 = j:
          if (w[j], X) in arc[a]:
            addToChart(X, i, j)
        for (B, X) in arc[a]:
          if B in chart[k+1, j]:
            addToChart(X, i, j)
    chart[i, j] = cube_prune(collections[i, j])

def addToChart(trie node X, int i, int j):
  if X has target collection:
    add X to collections[i, j]
  if arc[X] is not empty:
    add (X, j) to dotchart[i]
```

Algorithm 2: recursive CYK+

```
Input: array w of length N
initialize chart[N, N], collections[N, N]

root ← root node of grammar trie
for i in [N..1]:
  for j in [i..N]:

    if i = j:                                  #step 1
      if (w[i], X) in arc[root]:
        addToChart(X, i, j, false)
    else:                                      #step 2
      consume(root, i, i, j-1)
    chart[i, j] = cube_prune(collections[i, j])


def consume(trie node a, int i, int j, int k):
  unary ← i = j
  if j = k:
    if (w[j], X) in arc[a]:
      addToChart(X, i, k, unary)
  for (B, X) in arc[a]:
    if B in chart[j, k]:
      addToChart(X, i, k, unary)


def addToChart(trie node X, int i, int j, bool u):
  if X has target collection and u is false:
    add X to collections[i, j]
  if arc[X] is not empty:
    for k in [(j+1)..N]:
      consume(X, i, j+1, k)
```

Figure 2: side-by-side pseudocode of CYK+ (left) and our algorithm (right). Our algorithm uses a new chart traversal order and recursive *consume* function instead of a dot chart.

ant requires a right-to-left, depth-first chart traversal.

With our implementation of the SCFG as a trie, a type-2 is identified by a trie node, an array of back-pointers to antecedent cells, and a span. We distinguish between type-1 items before and after cube pruning. Productions, or specifically the target collections and back-pointers associated with them, are first added to a *collections* object, either synchronously or asynchronously. Cube pruning is always performed synchronously after all production of a cell have been found. Thus, the choice of algorithm does not change the search space in cube pruning, or the decoder output. After cube pruning, the chart cell is filled with a mapping from a non-terminal symbol to an object that compactly represents a collection of translation hypotheses and associated scores.

3.3 Chart Compression

Given a partial production for span (i, j), the number of chart cells in which the production can be continued is linear to sentence length. The recursive variant explicitly loops through all cells starting at position $j + 1$, but this search also exists in the original CYK+ in the form of the same type-2 item being re-used over time.

The guarantee that all cells $(j+1, k)$ are visited before cell (i, j) in the recursive algorithm allows for a further optimization. We construct a compressed matrix representation of the chart, which can be incrementally updated in O($|V| \cdot n^2$), V being the vocabulary of non-terminal symbols. For each start position and non-terminal symbol, we maintain an array of possible end positions and the corresponding chart entry, as illustrated in Table 1. The array is compressed in that it does not represent empty chart cells. Using the previous example, instead of searching all cells $(3, x)$ for a symbol NP, we only need to retrieve the array corresponding to start position 3 and symbol NP to obtain the array of cells which can continue the partial production.

While not affecting the time complexity of the algorithm, this compression technique reduces computational cost in two ways. If the chart is sparsely populated, i.e. if the size of the arrays is smaller than $n - j$, the algorithm iterates through fewer elements. Even if the chart is dense, we only perform one chart look-up per non-terminal and partial production, instead of $n - j$.

cell	S	NP	V	ART	NN
(3,3)				0x81	
(3,4)		0x86			

start	symbol	compressed column
3	ART	[(3, 0x81)]
3	NP	[(4, 0x86)]
3	S,V,NN	[]

Table 1: Matrix representation of all chart entries starting at position 3 (top), and equivalent compressed representation (bottom). Chart entries are pointers to objects that represent collection of translation hypotheses and their scores.

4 Related Work

Our proposed algorithm is similar to the work by Leermakers (1992), who describe a recursive variant of Earley's algorithm. While they discuss function memoization, which takes the place of charts in their work, as a space-time trade-off, a key insight of our work is that we can order the chart traversal in SCFG decoding so that partial productions need not be tabulated or memoized, without incurring any trade-off in time complexity.

Dunlop et al. (2010) employ a similar matrix compression strategy for CYK parsing, but their method is different to ours in that they employ matrix compression on the grammar, which they assume to be in Chomsky Normal Form, whereas we represent n-ary grammars as tries, and use matrix compression for the chart.

An obvious alternative to n-ary parsing is the use of binary grammars, and early SCFG models for SMT allowed only binary rules, as in the hierarchical models by Chiang (2007)[8], or binarizable ones as in inversion-transduction grammar (ITG) (Wu, 1997). Whether an n-ary rule can be binarized depends on the rule-internal reorderings between non-terminals; Zhang et al. (2006) describe a synchronous binarization algorithm.

Hopkins and Langmead (2010) show that the complexity of parsing n-ary rules is determined by the number of choice points, i.e. non-terminals that are initial, consecutive, or final, since terminal symbols in the rule constrain which cells are possible application contexts of a non-terminal symbol. They propose pruning of the SCFG to rules

[8]Specifically, Chiang (2007) allows at most two non-terminals per rule, and no adjacent non-terminals on the source side.

with at most 3 decision points, or scope 3, as an alternative to binarization that allows parsing in cubic time. In a runtime evaluation, SMT with their pruned, unbinarized grammar offers a better speed-quality trade-off than synchronous binarization because, even though both have the same complexity characteristics, synchronous binarization increases both the overall number of rules, and the number of non-terminals, which increases the grammar constant. In contrast, Chung et al. (2011) compare binarization and Earley-style parsing with scope-pruned grammars, and find Earley-style parsing to be slower. They attribute the comparative slowness of Earley-style parsing to the cost of building and storing the dot chart during decoding, which is exactly the problem that our paper addresses.

Williams and Koehn (2012) describe a parsing algorithm motivated by Hopkins and Langmead (2010) in which they store the grammar in a compact trie with source terminal symbols or a generic gap symbol as edge labels. Each path through this trie corresponds to a rule pattern, and is associated with the set of grammar rules that share the same rule pattern. Their algorithm initially constructs a secondary trie that records all rule patterns that apply to the input sentence, and stores the position of matching terminal symbols. Then, chart cells are populated by constructing a lattice for each rule pattern identified in the initial step, and traversing all paths through this lattice. Their algorithm is similar to ours in that they also avoid the construction of a dot chart, but they construct two other auxiliary structures instead: a secondary trie and a lattice for each rule pattern. In comparison, our algorithm is simpler, and we perform an empirical comparison of the two in the next section.

5 Empirical Results

We empirically compare our algorithm to the CYK+ algorithm, and the Scope-3 algorithm as described by Williams and Koehn (2012), in a string-to-tree SMT task. All parsing algorithms are equivalent in terms of translation output, and our evaluation focuses on memory consumption and speed.

5.1 Data

For SMT decoding, we use the Moses toolkit (Koehn et al., 2007) with KenLM for language model queries (Heafield, 2011). We use training data from the ACL 2014 Ninth Workshop on Statistical Machine Translation (WMT) shared translation task, consisting of 4.5 million sentence pairs of parallel data and a total of 120 million sentences of monolingual data. We build a string-to-tree translation system English→German, using target-side syntactic parses obtained with the dependency parser ParZu (Sennrich et al., 2013). A synchronous grammar is extracted with GHKM rule extraction (Galley et al., 2004; Galley et al., 2006), and the grammar is pruned to scope 3.

The synchronous grammar contains 38 million rule pairs with 23 million distinct source-side rules. We report decoding time for a random sample of 1000 sentences from the newstest2013/4 sets (average sentence length: 21.9 tokens), and peak memory consumption for sentences of 20, 40, and 80 tokens. We do not report the time and space required for loading the SMT models, which is stable for all experiments.[9] The parsing algorithm only accounts for part of the cost during decoding, and the relative gains from optimizing the parsing algorithm are highest if the rest of the decoder is fast. For best speed, we use cube pruning with language model boundary word grouping (Heafield et al., 2013) in all experiments. We set no limit to the maximal span of SCFG rules, but only keep the best 100 productions per span for cube pruning. The cube pruning limit itself is set to 1000.

algorithm	$n = 20$	$n = 40$	$n = 80$
Scope-3	0.02	0.04	0.34
CYK+	0.32	2.63	51.64
+ recursive	0.02	0.04	0.15
+ compression	0.02	0.04	0.15

Table 2: Peak memory consumption (in GB) of string-to-tree SMT decoder for sentences of different length n with different parsing algorithms.

5.2 Memory consumption

Peak memory consumption for different sentence lengths is shown in Table 2. For sentences of length 80, we observe more than 50 GB in peak memory consumption for CYK+, which makes it impractical for long sentences, especially for multi-threaded decoding. Our recursive variants keep memory consumption small, as does the

[9] The language model consumes 13 GB of memory, and the SCFG 37 GB. We leave the task of compacting the grammar to future research.

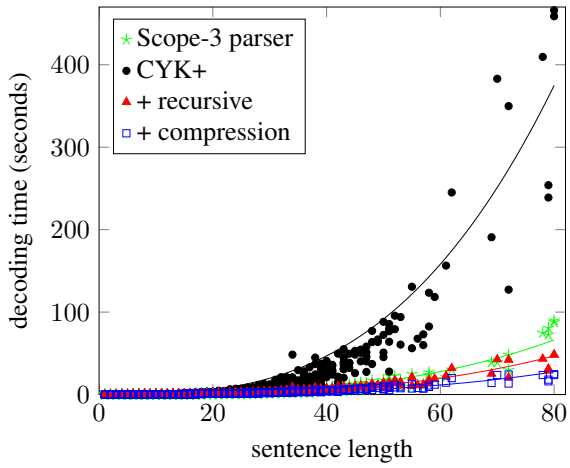

Figure 3: Decoding time per sentence as a function of sentence length for four parsing variants. Regression curves use least squares fitting on cubic function.

algorithm	length 80		random	
	parse	total	parse	total
Scope-3	74.5	81.1	1.9	2.6
CYK+	358.0	365.4	8.4	9.1
+ recursive	33.7	40.1	1.5	2.2
+ compression	15.0	21.2	1.0	1.7

Table 3: Parse time and total decoding time per sentence (in seconds) of string-to-tree SMT decoder with different parsing algorithms.

Scope-3 algorithm. This is in line with our theoretical expectation, since both algorithms eliminate the dot chart, which is the costliest data structure in the original CYK+ algorithm.

5.3 Speed

While the main motivation for eliminating the dot chart was to reduce memory consumption, we also find that our parsing variants are markedly faster than the original CYK+ algorithm. Figure 3 shows decoding time for sentences of different length with the four parsing variants. Table 3 shows selected results numerically, and also distinguishes between total decoding time and time spent in the parsing block, the latter ignoring the cost of cube pruning and language model scoring. If we consider parse time for sentences of length 80, we observe a speed-up by a factor of 24 between our fastest variant (with recursion and chart compression), and the original CYK+.

The gains from chart compression over the recursive variant – a factor 2 reduction in parse time for sentences of length 80 – are attributable to a reduction in the number of computational steps. The large speed difference between CYK+ and the recursive variant is somewhat more surprising, given the similarity of the two algorithms. Profiling results show that the recursive variant is not only faster because it saves the computational overhead of creating and destroying the dot chart, but that it also has a better locality of reference, with markedly fewer CPU cache misses.

Time differences are smaller for shorter sentences, both in terms of time spent parsing, and because the time spent outside of parsing is a higher proportion of the total. Still, we observe a factor 5 speed-up in total decoding time on our random translation sample from CYK+ to our fastest variant. We also observe speed-ups over the Scope-3 parser, ranging from a factor 5 speed-up (parsing time on sentences of length 80) to a 50% speed-up (total time on random translation sample). It is unclear to what extent these speed differences reflect the cost of building the auxiliary data structures in the Scope-3 parser, and how far they are due to implementation details.

5.4 Rule prefix scope

For the CYK+ parser, the growth of both memory consumption and decoding time exceeds our cubic growth expectation. We earlier remarked that the rule prefix of a scope-3 rule may actually be scope-4 if the prefix ends in a non-terminal, but the rule itself does not. Since this could increase space and time complexity of CYK+ to $O(n^4)$, we did additional experiments in which we prune all scope-3 rules with a scope-4 prefix. This affected 1% of all source-side rules in our model, and only had a small effect on translation quality (19.76 BLEU $\rightarrow$ 19.73 BLEU on newstest2013). With this additional pruning, memory consumption with CYK+ is closer to our theoretical expectation, with a peak memory consumption of 23 GB for sentences of length 80 ($\approx 2^3$ times more than for length 40). We also observe reductions in parse time as shown in Table 4. While we do see marked reductions in parse time for all CYK+ variants, our recursive variants maintain their efficiency advantage over the original algorithm. Rule prefix scope is irrelevant for the Scope-3 parsing algorithm[10], and its

[10]Despite its name, the Scope-3 parsing algorithm allows grammars of any scope, with a time complexity of $O(n^{scope(G)})$.

algorithm	length 80		random	
	full	pruned	full	pruned
Scope-3	74.5	70.1	1.9	1.8
CYK+	358.0	245.5	8.4	6.4
+ recursive	33.7	24.5	1.5	1.2
+ compression	15.0	10.5	1.0	0.8

Table 4: Average parse time (in seconds) of string-to-tree SMT decoder with different parsing algorithms, before and after scope-3 rules with scope-4 prefix have been pruned from grammar.

speed is only marginally affected by this pruning procedure.

6 Conclusion

While SCFG decoders with dot charts are still wide-spread, we argue that dot charts are only of limited use for SCFG decoding. The core contributions of this paper are the insight that a right-to-left, depth-first chart traversal order allows for the removal of the dot chart from the popular CYK+ algorithm without incurring any computational cost for SCFG decoding, and the presentation of a recursive CYK+ variant that is based on this insight. Apart from substantial savings in space complexity, we empirically demonstrate gains in decoding speed. The new chart traversal order also allows for a chart compression strategy that yields further speed gains.

Our parsing algorithm does not affect the search space or cause any loss in translation quality, and its speed improvements are orthogonal to improvements in cube pruning (Gesmundo et al., 2012; Heafield et al., 2013). The algorithmic modifications to CYK+ that we propose are simple, but we believe that the efficiency gains of our algorithm are of high practical importance for syntax-based SMT. An implementation of the algorithm has been released as part of the Moses SMT toolkit.

Acknowledgements

I thank Matt Post, Philip Williams, Marcin Junczys-Dowmunt and the anonymous reviewers for their helpful suggestions and feedback. This research was funded by the Swiss National Science Foundation under grant P2ZHP1_148717.

References

Jean-Cédric Chappelier and Martin Rajman. 1998. A Generalized CYK Algorithm for Parsing Stochastic CFG. In *TAPD*, pages 133–137.

David Chiang. 2007. Hierarchical Phrase-Based Translation. *Comput. Linguist.*, 33(2):201–228.

Tagyoung Chung, Licheng Fang, and Daniel Gildea. 2011. Issues Concerning Decoding with Synchronous Context-free Grammar. In *ACL (Short Papers)*, pages 413–417. The Association for Computer Linguistics.

Aaron Dunlop, Nathan Bodenstab, and Brian Roark. 2010. Reducing the grammar constant: an analysis of CYK parsing efficiency. Technical report CSLU-2010-02, OHSU.

Chris Dyer, Adam Lopez, Juri Ganitkevitch, Johnathan Weese, Ferhan Ture, Phil Blunsom, Hendra Setiawan, Vladimir Eidelman, and Philip Resnik. 2010. cdec: A Decoder, Alignment, and Learning framework for finite-state and context-free translation models. In *Proceedings of the Association for Computational Linguistics (ACL)*.

Michel Galley, Mark Hopkins, Kevin Knight, and Daniel Marcu. 2004. What's in a Translation Rule? In *HLT-NAACL '04*.

Michel Galley, Jonathan Graehl, Kevin Knight, Daniel Marcu, Steve DeNeefe, Wei Wang, and Ignacio Thayer. 2006. Scalable inference and training of context-rich syntactic translation models. In *ACL-44: Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 961–968, Sydney, Australia. Association for Computational Linguistics.

Andrea Gesmundo, Giorgio Satta, and James Henderson. 2012. Heuristic Cube Pruning in Linear Time. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers - Volume 2*, ACL '12, pages 296–300, Jeju Island, Korea. Association for Computational Linguistics.

Kenneth Heafield, Philipp Koehn, and Alon Lavie. 2013. Grouping Language Model Boundary Words to Speed K-Best Extraction from Hypergraphs. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 958–968, Atlanta, Georgia, USA.

Kenneth Heafield. 2011. KenLM: Faster and Smaller Language Model Queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, Edinburgh, UK. Association for Computational Linguistics.

Mark Hopkins and Greg Langmead. 2010. SCFG Decoding Without Binarization. In *EMNLP*, pages 646–655.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open Source Toolkit for Statistical Machine Translation. In *Proceedings of the ACL-2007 Demo and Poster Sessions*, pages 177–180, Prague, Czech Republic. Association for Computational Linguistics.

René Leermakers. 1992. A recursive ascent Earley parser. *Information Processing Letters*, 41(2):87–91, February.

Rico Sennrich, Martin Volk, and Gerold Schneider. 2013. Exploiting Synergies Between Open Resources for German Dependency Parsing, POS-tagging, and Morphological Analysis. In *Proceedings of the International Conference Recent Advances in Natural Language Processing 2013*, pages 601–609, Hissar, Bulgaria.

Ashish Venugopal and Andreas Zollmann. 2009. Grammar based statistical MT on Hadoop: An end-to-end toolkit for large scale PSCFG based MT. *The Prague Bulletin of Mathematical Linguistics*, 91:67–78.

David Vilar, Daniel Stein, Matthias Huck, and Hermann Ney. 2012. Jane: an advanced freely available hierarchical machine translation toolkit. *Machine Translation*, 26(3):197–216.

Philip Williams and Philipp Koehn. 2012. GHKM Rule Extraction and Scope-3 Parsing in Moses. In *Proceedings of the Seventh Workshop on Statistical Machine Translation*, pages 388–394, Montréal, Canada, June. Association for Computational Linguistics.

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

Hao Zhang, Liang Huang, Daniel Gildea, and Kevin Knight. 2006. Synchronous Binarization for Machine Translation. In *HLT-NAACL*. The Association for Computational Linguistics.