# Faster MT Decoding through Pervasive Laziness

**Michael Pust and Kevin Knight**
Information Sciences Institute
University of Southern California
*lastname*@isi.edu

## Abstract

Syntax-based MT systems have proven effective—the models are compelling and show good room for improvement. However, decoding involves a slow search. We present a new lazy-search method that obtains significant speedups over a strong baseline, with no loss in Bleu.

## 1 Introduction

Syntax-based string-to-tree MT systems have proven effective—the models are compelling and show good room for improvement. However, slow decoding hinders research, as most experiments involve heavy parameter tuning, which involves heavy decoding. In this paper, we present a new method to improve decoding performance, obtaining a significant speedup over a strong baseline with no loss in Bleu. In scenarios where fast decoding is more important than optimal Bleu, we obtain better Bleu for the same time investment. Our baseline is a full-scale syntax-based MT system with 245m tree-transducer rules of the kind described in (Galley et al., 2004), 192 English non-terminal symbols, an integrated 5-gram language model (LM), and a decoder that uses state-of-the-art cube pruning (Chiang, 2007). A sample translation rule is:

$$S(x0{:}NP\ x1{:}VP) \leftrightarrow x1{:}VP\ x0{:}NP$$

In CKY string-to-tree decoding, we attack spans of the input string from shortest to longest. We populate each span with a set of edges. An edge contains a English non-terminal (NT) symbol (NP, VP, etc), border words for LM combination, pointers to child edges, and a score. The score is a sum of (1) the left-child edge score, (2) the right-child edge score, (3) the score of the translation rule that combined them, and (4) the target-string LM score. In this paper, we are only concerned with what happens when constructing edges for a single span [i,j]. The naive algorithm works like this:

```
for each split point k
  for each edge A in span [i,k]
    for each edge B in span [k,j]
      for each rule R with RHS = A B
        create new edge for span [i,j]
  delete all but 1000-best edges
```

The last step provides a necessary beam. Without it, edges proliferate beyond available memory and time. But even with the beam, the naive algorithm fails, because enumerating all <A,B,R> triples at each span is too time consuming.

## 2 Cube Pruning

Cube pruning (Chiang, 2007) solves this problem by lazily enumerating triples. To work, cube pruning requires that certain orderings be continually maintained at all spans. First, rules are grouped by RHS into *rule sets* (eg, all the NP-VP rules are in a set), and the members of a given set are sorted by rule score. Second, edges in a span are grouped by NT into *edge sets* (eg, all the NP edges are in an edge set), ordered by edge score.

Consider the sub-problem of building new [i,j] edges by combining (just) the NP edges over [i,k] with (just) the VP edges over [k,j], using the available NP-VP rules. Rather than enumerate all triples, cube pruning sets up a 3-dimensional cube structure whose individually-sorted axes are the NP left edges, the VP right edges, and the NP-VP rules. Because the corner of the cube (best NP left-edge, best VP right-edge, best NP-VP rule) is likely the best edge in the cube, at beam size 1, we would simply return this edge and terminate, without checking other triples. We say "likely" because the corner position does not take into account the LM portion of the score.[1]

After we take the corner and post a new edge from it, we identify its 3 neighbors in the cube. We com-

---

[1] We also employ LM rule and edge forward-heuristics as in (Chiang, 2007), which improve the sorting.

pute their full scores (including LM portion) and push them onto a priority queue (PQ). We then pop an item from the PQ, post another new edge, and push the item's neighbors onto the PQ. Note that this PQ grows in size over time. In this way, we explore the best portion of the cube without enumerating all its contents. Here is the algorithm:

    push(corner, make-edge(corner)) onto PQ
    for i = 1 to 1000
        pop(position, edge) from top of PQ
        post edge to chart
        for each n in neighbors(position)
          push(n, make-edge(n)) onto PQ
        if PQ is empty, break from for-loop

The function *make-edge* completely scores an edge (including LM score) before inserting it into the PQ. Note that in practice, we execute the loop up to 10k times, to get 1000 edges that are distinct in their NTs and border words.

In reality, we have to construct many cubes, one for each combinable left and right edge set for a given split point, plus all the cubes for all the other split points. So we maintain a PQ-of-PQs whose elements are cubes.

    create each cube, pushing its fully-scored corner
      onto the cube's PQ
    push cubes themselves onto a PQ-of-PQs
    for i = 1 to 1000:
        pop a cube C from the PQ-of-PQs
        pop an item from C
          post edge to chart
          retrieve neighbors, score & push them onto C
        push C back onto the PQ-of-PQs

## 3   Lazy Lists

When we meter the cube pruning algorithm, we find that over 80% of the time goes to building the initial queue of cubes, including deriving a corner edge for each cube—only a small fraction is spent deriving additional edges via exploring the cubes. For spans of length 10 or greater, we find that we have to create more than 1000 cubes, i.e., more than the number of edges we wish to explore.

Our idea, then, is to create the cubes themselves lazily. To describe our algorithm, we exploit an abstract data structure called a *lazy list* (aka generator, stream, pipe, or iterator), which supports three oper-

ations:

    next(list): pops the front item from a list
    peek(list): returns the score of the front item
    empty(list): returns true if the list is empty

A cube is a lazy list (of edges). For our purposes, a lazy list can be implemented with a PQ or something else—we no longer care how the list is populated or maintained, or even whether there are a finite number of elements.

Instead of explicitly enumerating all cubes for a span, we aim to produce a lazy list of cubes. Assume for the moment that such a lazy list exists—we show how to create it in the next section—and call it L. Let us also say that cubes come off L in order of their top edges' scores. To get our first edge, we let C = next(L), and then we call next(C). Now a question arises: do we pop the next-best edge off C, or do we investigate the next cube in L? We can decide by calling peek(peek(L)). If we choose to pop the next cube (and then its top edge), then we face another (this time three-way) decision. Bookkeeping is therefore required if we are to continue to emit edges in a good order.

We manage the complexity through the abstraction of a *lazy list of lazy lists*, to which we routinely apply a single, key operation called *merge-lists*. This operation converts a lazy list of lazy lists of X's into a simple lazy list of X's. X can be anything: edges, integers, lists, lazy lists, etc.

Figure 1 gives the generic merge-lists algorithm. The *yield* function suspends computation and returns to the caller. peek() lets the caller see what is yielded, next() returns what is yielded and resumes the loop, and empty() tells if the loop is still active.

We are now free to construct any nested "list of lists of lists ... of lists of X" (all lazy) and reduce it stepwise and automatically to a single lazy list. Standard cube pruning (Section 2) provides a simple example: if L is a list of cubes, and each cube is a lazy list of edges, then merge-lists(L) returns us a lazy list of edges (M), which is exactly what the decoder wants. The decoder can populate a new span by simply making 1000 calls to next(M).

## 4   Pervasive Laziness

Now we describe how to generate cubes lazily. As with standard cube pruning, we need to maintain a

merge-lists(L):
 (L is a lazy list of lazy lists)
1. set up an empty PQ of lists,
   prioritized by peek(list)
2. push next(L) onto PQ
3. pop list L2 off PQ
4. yield pop(L2)
5. if !empty(L2) and peek(L2) is worse than
   peek(peek(L)), then push next(L) onto PQ
6. if !empty(L2), then push L2 onto PQ
7. go to step 3

Figure 1: Generic merge-lists algorithm.



Figure 2: Organizing lazy lists for the decoder.

small amount of ordering information among edges in a span, which we exploit in constructing higher-level spans. Previously, we required that all NP edges be ordered by score, the same for VP edges, etc. Now we additionally order whole *edge sets* (groups of edges sharing an NT) with respect to each other, eg, NP > VP > RB > etc. These are ordered by the top-scoring edges in each set.

Ideally, we would pop cubes off our lazy list in order of their top edges. Recall that the PQ-of-PQs in standard cube pruning works this way. We cannot guarantee this anymore, so we approximate it.

Consider first a single edge set from [i,k], eg, all the NP edges. We build a lazy list of cubes that all have a left-NP. Because edge sets from [k,j] are ordered with respect to each other, we may find that it is the VP edge set that contains the best edge in [k,j]. Pulling in all NP-VP rules, we can now postulate a "best cube," which generates edges out of left-NPs and right-VPs. We can either continue making edge from this cube, or we can ask for a "second-best cube" by moving to the next edge set of [k,j], which might contain all the right-PP edges. Thus, we have a lazy list of left-NP cubes. Its ordering is approximate—cubes come off in such a way that their top edges go from best to worst, but only considering the left and right child scores, not the rule scores. This is the same idea followed by standard cube pruning when it ignores internal LM scores.

We next create similar lazy lists for all the other [i,k] edge sets (not just NP). We combine these lists into a higher-level lazy list, whose elements pop off according to the ordering of edge sets in [i,k]. This structure contains all edges that can be produced
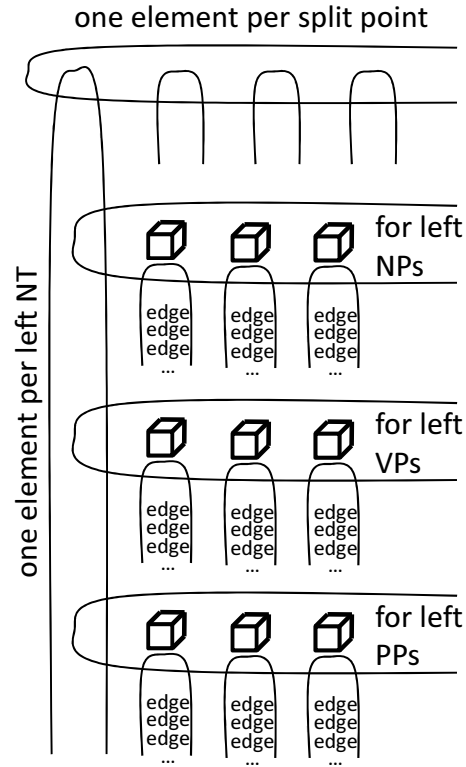
from split point k. We call merge-lists recursively on the structure, leaving us with a single lazy list M of edges. The decoder can now make 1000 calls to next(M) to populate the new span.

Edges from other split points, however, must compete on an equal basis for those 1000 slots. We therefore produce a separate lazy list for each of the $j - i - 1$ split points and combine these into an even higher-level list. Lacking an ordering criterion among split points, we presently make the top list a non-lazy one via the PQ-of-PQs structure. Figure 2 shows how our lists are organized.

The quality of our 1000-best edges can be improved. When we organize the higher-level lists by left edge-sets, we give prominence to the best left edge-set (eg, NP) over others (eg, VP). If the left span is relatively short, the contribution of the left NP to the total score of the new edge is small, so this prominence is misplaced. Therefore, we repeat the above process with the higher-level lists organized by right span instead of left. We merge the right-oriented and left-oriented structures, making sure that duplicates are avoided.
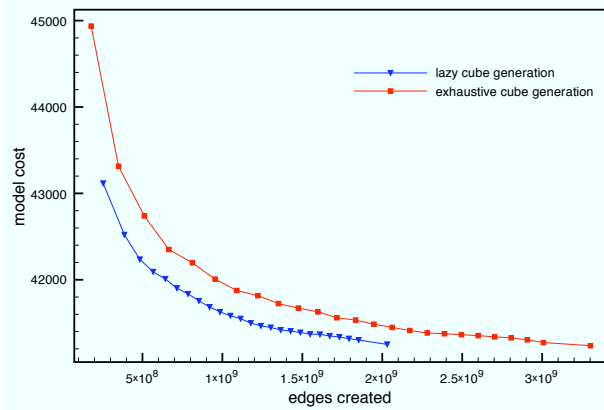
**Related Work.** Huang and Chiang (2007) de-

Figure 3: Number of edges produced by the decoder, versus model cost of 1-best decodings.



Figure 4: Decoding time versus Bleu.

scribe a variation of cube pruning called cube growing, and they apply it to a source-tree to target-string translator. It is a two pass approach, where a context-free parser is used to build a source forest, and a top down lazy forest expansion is used to integrate a language model. The expansion recursively calls cubes top-down, in depth first order. The context-free forest controls which cubes are built, and acts as a heuristic to minimize the number of items returned from each cube necessary to generate k-best derivations at the top.

It is not clear that a decoder such as ours, without the source-tree constraint, would benefit from this method, as building a context-free forest consistent with future language model integration via cubes is expensive on its own. However, we see potential integration of both methods in two places: First, the merge-lists algorithm can be used to lazily process any nested for-loops—including vanilla CKY—provided the iterands of the loops can be prioritized. This could speed up the creation of a first-pass context-free forest. Second, the cubes themselves could be prioritized in a manner similar to what we describe, using the context-free forest to prioritize cube generation rather than antecedent edges in the chart (since those do not exist yet).

## 5   Results

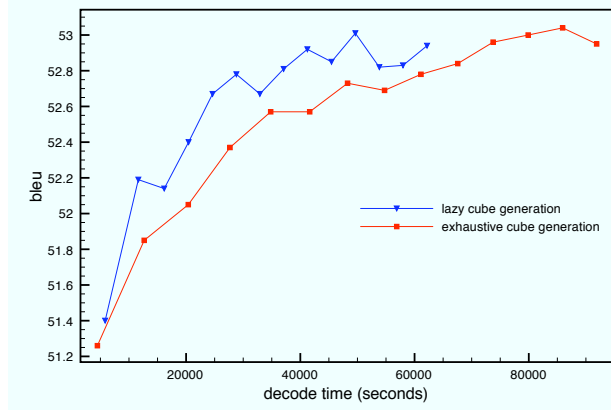We compare our method with standard cube pruning (Chiang, 2007) on a full-scale Arabic/English syntax-based MT system with an integrated 5-gram LM. We report on 500 test sentences of lengths 15-35. There are three variables of interest: runtime, model cost (summed across all sentences), and IBM Bleu. By varying the beam sizes (up to 1350), we obtain curves that plot edges-produced versus model-cost, shown in Figure 3. Figure 4 plots Bleu score against time. We see that we have improved the way our decoder searches, by teaching it to explore fewer edges, without sacrificing its ability to find low-cost edges. This leads to faster decoding without loss in translation accuracy.

Taken together with cube pruning (Chiang, 2007), k-best tree extraction (Huang and Chiang, 2005), and cube growing (Huang and Chiang, 2007), these results provide evidence that lazy techniques may penetrate deeper yet into MT decoding and other NLP search problems.

## References

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

M. Galley, M. Hopkins, K. Knight, and D. Marcu. 2004. What's in a translation rule. In *Proc. NAACL-HLT*.

L. Huang and D. Chiang. 2005. Better k-best parsing. In *Proc. IWPT*.

L. Huang and D. Chiang. 2007. Forest rescoring: Faster decoding with integrated language models. In *Proc. ACL*.