

Cube Pruning as Heuristic Search

Mark Hopkins and Greg Langmead

Language Weaver, Inc.

4640 Admiralty Way, Suite 1210

Marina del Rey, CA 90292

{mhopkins, glangmead}@languageweaver.com

Abstract

Cube pruning is a fast inexact method for generating the items of a beam decoder. In this paper, we show that cube pruning is essentially equivalent to A^* search on a specific search space with specific heuristics. We use this insight to develop faster and exact variants of cube pruning.

1 Introduction

In recent years, an intense research focus on machine translation (MT) has raised the quality of MT systems to the degree that they are now viable for a variety of real-world applications. Because of this, the research community has turned its attention to a major drawback of such systems: they are still quite slow. Recent years have seen a flurry of innovative techniques designed to tackle this problem. These include cube pruning (Chiang, 2007), cube growing (Huang and Chiang, 2007), early pruning (Moore and Quirk, 2007), closing spans (Roark and Hollingshead, 2008; Roark and Hollingshead, 2009), coarse-to-fine methods (Petrov et al., 2008), pervasive laziness (Pust and Knight, 2009), and many more.

This massive interest in speed is bringing rapid progress to the field, but it comes with a certain amount of baggage. Each technique brings its own terminology (from the *cubes* of (Chiang, 2007) to the *lazy lists* of (Pust and Knight, 2009)) into the mix. Often, it is not entirely clear why they work. Many apply only to specialized MT situations. Without a deeper understanding of these methods, it is difficult for the practitioner to combine them and adapt them to new use cases.

In this paper, we attempt to bring some clarity to the situation by taking a closer look at one of these existing methods. Specifically, we cast the popular technique of *cube pruning* (Chiang, 2007) in the well-understood terms of heuristic search

(Pearl, 1984). We show that cube pruning is essentially equivalent to A^* search on a specific search space with specific heuristics. This simple observation affords a deeper insight into how and why cube pruning works. We show how this insight enables us to easily develop faster and exact variants of cube pruning for tree-to-string transducer-based MT (Galley et al., 2004; Galley et al., 2006; DeNero et al., 2009).

2 Motivating Example

We begin by describing the problem that cube pruning addresses. Consider a synchronous context-free grammar (SCFG) that includes the following rules:

$$A \rightarrow \langle A_{\square} B_{\square}, A_{\square} B_{\square} \rangle \quad (1)$$

$$B \rightarrow \langle A_{\square} B_{\square}, B_{\square} A_{\square} \rangle \quad (2)$$

$$A \rightarrow \langle B_{\square} A_{\square}, c B_{\square} b A_{\square} \rangle \quad (3)$$

$$B \rightarrow \langle B_{\square} A_{\square}, B_{\square} A_{\square} \rangle \quad (4)$$

Figure 1 shows CKY decoding in progress. CKY is a bottom-up algorithm that works by building objects known as *items*, over increasingly larger *spans* of an input sentence (in the context of SCFG decoding, the items represent partial translations of the input sentence). To limit running time, it is common practice to keep only the n “best” items per span (this is known as *beam decoding*). At this point in Figure 1, every span of size 2 or less has already been filled, and now we want to fill span $[2, 5]$ with the n items of lowest cost. Cube pruning addresses the problem of how to compute the n -best items efficiently.

We can be more precise if we introduce some terminology. An SCFG *rule* has the form $X \rightarrow \langle \sigma, \varphi, \sim \rangle$, where X is a nonterminal (called the *postcondition*), σ, φ are strings that may contain terminals and nonterminals, and \sim is a 1-1 correspondence between equivalent nonterminals of σ and φ .

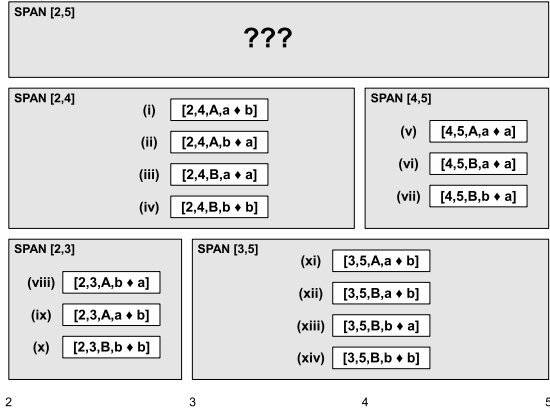


Figure 1: CKY decoding in progress. We want to fill span [2,5] with the lowest cost items.

Usually SCFG rules are represented like the example rules (1)-(4). The subscripts indicate corresponding nonterminals (according to \sim). Define the *preconditions* of a rule as the ordered sequence of its nonterminals. For clarity of presentation, we will henceforth restrict our focus to binary rules, i.e. rules of the form: $Z \rightarrow \langle X_{\square} Y_{\square}, \varphi \rangle$. Observe that all the rules of our example are binary rules.

An *item* is a triple that contains a span and two strings. We refer to these strings as the *postcondition* and the *carry*, respectively. The *postcondition* tells us which rules may be applied to the item. The *carry* gives us extra information required to correctly score the item (in SCFG decoding, typically it consists of boundary words for an n -gram language model).¹ To flatten the notation, we will generally represent items as a 4-tuple, e.g. $[2, 4, X, a \diamond b]$.

In CKY, new items are created by applying rules to existing items:

$$\frac{r : Z \rightarrow \langle X_{\square} Y_{\square}, \varphi \rangle \quad [\alpha, \delta, X, \kappa_1] \quad [\delta, \beta, Y, \kappa_2]}{[\alpha, \beta, Z, \text{carry}(r, \kappa_1, \kappa_2)]} \quad (5)$$

In other words, we are allowed to apply a rule r to a pair of items ι_1, ι_2 if the item spans are complementary and $\text{preconditions}(r) = \langle \text{postcondition}(\iota_1), \text{postcondition}(\iota_2) \rangle$. The new item has the same postcondition as the applied rule. We form the carry for the new item through an application-dependent function carry that combines the carries of its subitems (e.g. if the carry is n -gram boundary words, then carry computes the

¹Note that the carry is a generic concept that can store any kind of non-local scoring information.

new boundary words). As a shorthand, we introduce the notation $\iota_1 \triangleright r \triangleleft \iota_2$ to describe an item created by applying formula (5) to rule r and items ι_1, ι_2 .

When we create a new item, it is scored using the following formula:²

$$\begin{aligned} \text{cost}(\iota_1 \triangleright r \triangleleft \iota_2) \triangleq & \text{cost}(r) \\ & + \text{cost}(\iota_1) \\ & + \text{cost}(\iota_2) \\ & + \text{interaction}(r, \kappa_1, \kappa_2) \end{aligned} \quad (6)$$

We assume that each grammar rule r has an associated cost, denoted $\text{cost}(r)$. The *interaction cost*, denoted $\text{interaction}(r, \kappa_1, \kappa_2)$, uses the carry information to compute cost components that cannot be incorporated offline into the rule costs (again, for our purposes, this is a language model score).

Cube pruning addresses the problem of efficiently computing the n items of lowest cost for a given span.

3 Item Generation as Heuristic Search

Refer again to the example in Figure 1. We want to fill span [2,5]. There are 26 distinct ways to apply formula (5), which result in 10 unique items. One approach to finding the lowest-cost n items: perform all 26 distinct inferences, compute the cost of the 10 unique items created, then choose the lowest n .

The 26 different ways to form the items can be structured as a search tree. See Figure 2. First we choose the subspans, then the rule preconditions, then the rule, and finally the subitems. Notice that this search space is already quite large, even for such a simple example. In a realistic situation, we are likely to have a search tree with thousands (possibly millions) of nodes, and we may only want to find the best 100 or so goal nodes. To explore this entire search space seems wasteful. Can we do better?

Why not perform heuristic search directly on this search space to find the lowest-cost n items? In order to do this, we just need to add heuristics to the internal nodes of the space.

Before doing so, it will help to elaborate on some of the details of the search tree. Let $\text{rules}(X, Y)$ be the subset of rules with preconditions $\langle X, Y \rangle$, sorted by increasing cost. Similarly,

²Without loss of generality, we assume an additive cost function.

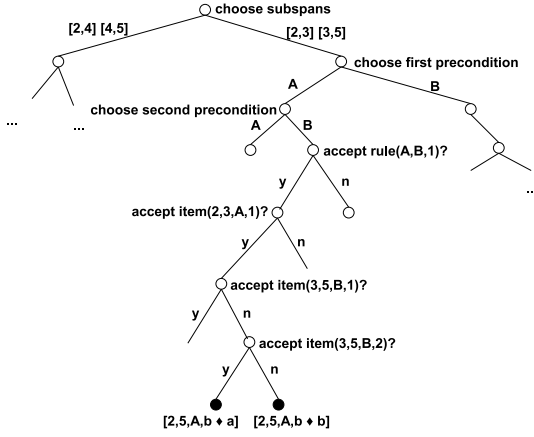


Figure 2: Item creation, structured as a search space. $\text{rule}(X, Y, k)$ denotes the k^{th} lowest-cost rule with preconditions $\langle X, Y \rangle$. $\text{item}(\alpha, \beta, X, k)$ denotes the k^{th} lowest-cost item of span $[\alpha, \beta]$ with postcondition X .

let $\text{items}(\alpha, \beta, X)$ be the subset of items with span $[\alpha, \beta]$ and postcondition X , also sorted by increasing cost. Finally, let $\text{rule}(X, Y, k)$ denote the k^{th} rule of $\text{rules}(X, Y)$ and let $\text{item}(\alpha, \beta, X, k)$ denote the k^{th} item of $\text{items}(\alpha, \beta, X)$.

A path through the search tree consists of the following sequence of decisions:

1. Set i, j, k to 1.
2. Choose the subspans: $[\alpha, \delta], [\delta, \beta]$.
3. Choose the first precondition X of the rule.
4. Choose the second precondition Y of the rule.
5. While rule not yet accepted and $i < |\text{rules}(X, Y)|$:
 - (a) Choose to accept/reject $\text{rule}(X, Y, i)$. If reject, then increment i .
6. While item not yet accepted for subspan $[\alpha, \delta]$ and $j < |\text{items}(\alpha, \delta, X)|$:
 - (a) Choose to accept/reject $\text{item}(\alpha, \delta, X, j)$. If reject, then increment j .
7. While item not yet accepted for subspan $[\delta, \beta]$ and $k < |\text{items}(\delta, \beta, Y)|$:
 - (a) Choose to accept/reject $\text{item}(\delta, \beta, Y, k)$. If reject, then increment k .

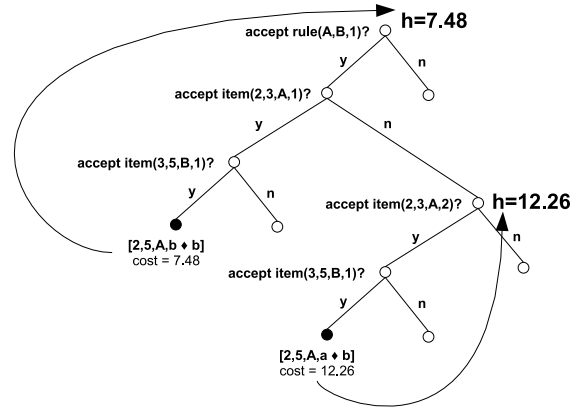


Figure 3: The lookahead heuristic. We set the heuristics for rule and item nodes by looking ahead at the cost of the greedy solution from that point in the search space.

Figure 2 shows two complete search paths for our example, terminated by *goal nodes* (in black). Notice that the internal nodes of the search space can be classified by the type of decision they govern. To distinguish between these nodes, we will refer to them as *subspan nodes*, *precondition nodes*, *rule nodes*, and *item nodes*.

We can now proceed to attach heuristics to the nodes and run a heuristic search protocol, say A^* , on this search space. For subspan and precondition nodes, we attach trivial uninformative heuristics, i.e. $h = -\infty$. For goal nodes, the heuristic is the actual cost of the item they represent. For rule and item nodes, we will use a simple type of heuristic, often referred to in the literature as a *lookahead heuristic*. Since the rule nodes and item nodes are ordered, respectively, by rule and item cost, it is possible to “look ahead” at a greedy solution from any of those nodes. See Figure 3. This greedy solution is reached by choosing to accept every decision presented until we hit a goal node.

If these heuristics were admissible (i.e. lower bounds on the cost of the best reachable goal node), this would enable us to exactly generate the n -best items without exhausting the search space (assuming the heuristics are strong enough for A^* to do some pruning). Here, the lookahead heuristics are clearly not admissible, however the hope is that A^* will generate n “good” items, and that the time savings will be worth sacrificing exactness for.

4 Cube Pruning as Heuristic Search

In this section, we will compare cube pruning with our A* search protocol, by tracing through their respective behaviors on the simple example of Figure 1.

4.1 Phase 1: Initialization

To fill span $[\alpha, \beta]$, cube pruning (CP) begins by constructing a *cube* for each tuple of the form:

$$\langle [\alpha, \delta], [\delta, \beta], X, Y \rangle$$

where X and Y are nonterminals. A cube consists of three axes: rules(X, Y) and items(α, δ, X) and items(δ, β, Y). Figure 4(left) shows the nontrivial cubes for our example scenario.

Contrast this with A*, which begins by adding the root node of our search space to an empty heap (ordered by heuristic cost). It proceeds to repeatedly pop the lowest-cost node from the heap, then add its children to the heap (we refer to this operation as *visiting* the node). Note that before A* ever visits a rule node, it will have visited every subspan and precondition node (because they all have cost $h = -\infty$). Figure 4(right) shows the state of A* at this point in the search. We assume that we do not generate dead-end nodes (a simple matter of checking that there exist applicable rules and items for the chosen subspans and preconditions). Observe the correspondence between the cubes and the heap contents at this point in the A* search.

4.2 Phase 2: Seeding the Heap

Cube pruning proceeds by computing the “best” item of each cube $\langle [\alpha, \delta], [\delta, \beta], X, Y \rangle$, i.e.

$$\text{item}(\alpha, \delta, X, 1) \succ \text{rule}(X, Y, 1) \prec \text{item}(\delta, \beta, Y, 1)$$

Because of the interaction cost, there is no guarantee that this will really be the best item of the cube, however it is likely to be a good item because the costs of the individual components are low. These items are added to a heap (to avoid confusion, we will henceforth refer to the two heaps as the *CP heap* and the *A* heap*), and prioritized by their costs.

Consider again the example. CP seeds its heap with the “best” items of the 4 cubes. There is now a direct correspondence between the CP heap and the A* heap. Moreover, the costs associated with the heap elements also correspond. See Figure 5.

4.3 Phase 3: Finding the First Item

Cube pruning now pops the lowest-cost item from the CP heap. This means that CP has decided to keep the item. After doing so, it forms the “one-off” items and pushes those onto the CP heap. See Figure 5(left). The popped item is:

$$\text{item (viii)} \succ \text{rule (1)} \prec \text{item (xii)}$$

CP then pushes the following one-off successors onto the CP heap:

$$\text{item (viii)} \succ \text{rule (2)} \prec \text{item (xii)}$$

$$\text{item (ix)} \succ \text{rule (1)} \prec \text{item (xii)}$$

$$\text{item (viii)} \succ \text{rule (1)} \prec \text{item (xiii)}$$

Contrast this with A*, which pops the lowest-cost *search node* from the A* heap. Here we need to assume that our A* protocol differs slightly from standard A*. Specifically, it will practice *node-tying*, meaning that when it visits a rule node or an item node, then it also (atomically) visits all nodes on the path to its lookahead goal node. See Figure 5(right). Observe that all of these nodes have the same heuristic cost, thus standard A* is *likely* to visit these nodes in succession without the need to enforce node-tying, but it would not be guaranteed (because the heuristics are not admissible). A* keeps the goal node it finds and adds the successors to the heap, scored with their lookahead heuristics. Again, note the direct correspondence between what CP and A* keep, and what they add to their respective heaps.

4.4 Phase 4: Finding Subsequent Items

Cube pruning and A* continue to repeat Phase 3 until k unique items have been kept. While we could continue to trace through the example, by now it should be clear: cube pruning and our A* protocol with node-tying are doing the same thing at each step. In fact, they are *exactly the same algorithm*. We do not present a formal proof here; this statement should be regarded as confident conjecture.

The node-tying turns out to be an unnecessary artifact. In our early experiments, we discovered that node-tying has no impact on speed or quality. Hence, for the remainder of the paper, we view cube pruning in very simple terms: as nothing more than standard A* search on the search space of Section 3.

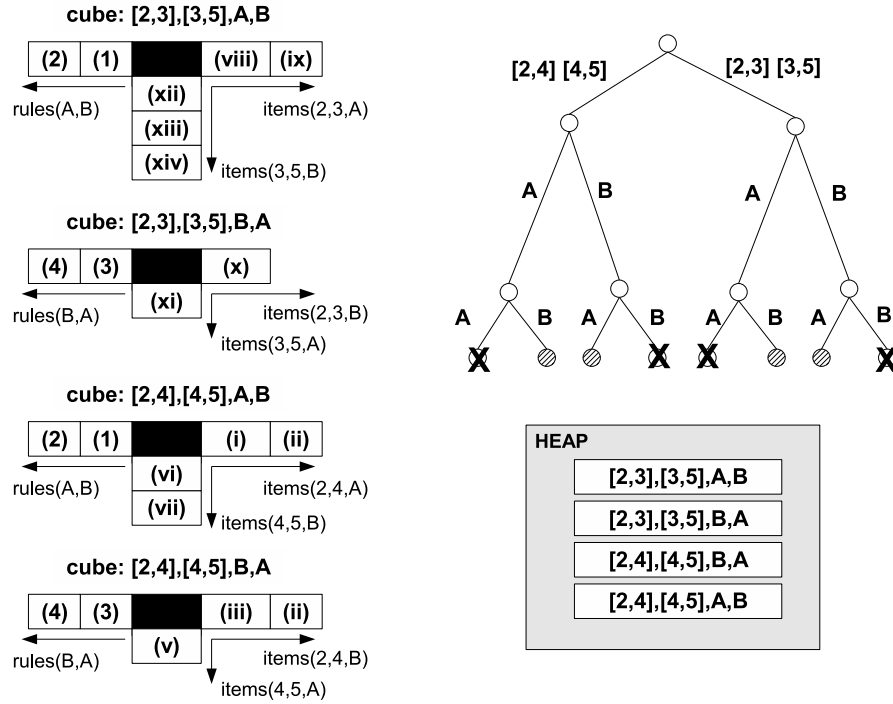


Figure 4: (left) Cube formation for our example. (right) The A* protocol, after all subspan and precondition nodes have been visited. Notice the correspondence between the cubes and the A* heap contents.

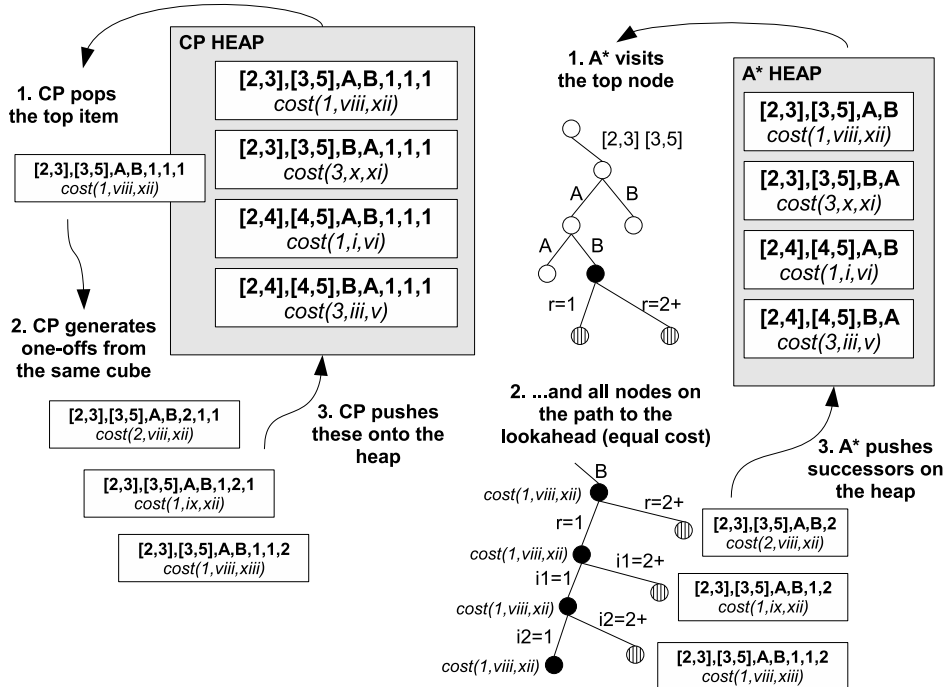


Figure 5: (left) One step of cube pruning. (right) One step of the A* protocol. In this figure, $\text{cost}(r, l_1, l_2) \triangleq \text{cost}(l_1 > r < l_2)$.

5 Augmented Cube Pruning

Viewed in this light, the idiosyncracies of cube pruning begin to reveal themselves. On the one hand, rule and item nodes are associated with strong but inadmissible heuristics (the short explanation for why cube pruning is an inexact algorithm). On the other hand, subspan and precondition nodes are associated with weak trivial heuristics. This should be regarded neither as a surprise nor a criticism, considering cube pruning’s origins in hierarchical phrase-based MT models (Chiang, 2007), which have only a small number of distinct nonterminals.

But the situation is much different in tree-to-string transducer-based MT (Galley et al., 2004; Galley et al., 2006; DeNero et al., 2009). Transducer-based MT relies on SCFGs with large nonterminal sets. Binarizing the grammars (Zhang et al., 2006) further increases the size of these sets, due to the introduction of virtual nonterminals.

A key benefit of the heuristic search viewpoint is that it is well positioned to take advantage of such insights into the structure of a particular decoding problem. In the case of transducer-based MT, the large set of preconditions encourages us to introduce a nontrivial heuristic for the precondition nodes. The inclusion of these heuristics into the CP search will enable A* to eliminate certain preconditions from consideration, giving us a speedup. For this reason we call this strategy *augmented cube pruning*.

5.1 Heuristics on preconditions

Recall that the total cost of a goal node is given by Equation (6), which has four terms. We will form the heuristic for a precondition node by creating a separate heuristic for each of the four terms and using the sum as the overall heuristic.

To describe these heuristics, we will make intuitive use of the wildcard operator $*$ to extend our existing notation. For instance, $\text{items}(\alpha, \beta, *)$ will denote the union of $\text{items}(\alpha, \beta, X)$ over all possible X , sorted by cost.

We associate the heuristic $h(\delta, X, Y)$ with the search node reached by choosing subspans $[\alpha, \delta]$, $[\delta, \beta]$, precondition X (for span $[\alpha, \delta]$), and precondition Y (for span $[\delta, \beta]$). The heuristic is the sum of four terms, mirroring Equation (6):

$$\begin{aligned} h(\delta, X, Y) &= \text{cost}(\text{rule}(X, Y, 1)) \\ &+ \text{cost}(\text{item}(\alpha, \delta, X, 1)) \\ &+ \text{cost}(\text{item}(\delta, \beta, Y, 1)) \\ &+ \text{ih}(\delta, X, Y) \end{aligned}$$

The first three terms are admissible because each is simply the minimum possible cost of some choice remaining to be made. To construct the *interaction heuristic* $\text{ih}(\delta, X, Y)$, consider that in a translation model with an integrated n -gram language model, the interaction cost $\text{interaction}(r, \kappa_1, \kappa_2)$ is computed by adding the language model costs of any new complete n -grams that are created by combining the carries (boundary words) with each other and with the lexical items on the rule’s target side, taking into account any reordering that the rule may perform. We construct a backoff-style estimate of these new n -grams by looking at $\text{item}(\alpha, \delta, X, 1) = [\alpha, \delta, X, \kappa_1]$, $\text{item}(\delta, \beta, Y, 1) = [\delta, \beta, Y, \kappa_2]$, and $\text{rule}(X, Y, 1)$. We set $\text{ih}(\delta, X, Y)$ to be a linear combination of the backoff n -grams of the carries κ_1 and κ_2 , as well as any n -grams introduced by the rule. For instance, if

$$\kappa_1 = \text{a b} \diamond \text{c d}$$

$$\kappa_2 = \text{e f} \diamond \text{g h}$$

$$\text{rule}(X, Y, 1) = Z \rightarrow \langle X_{\square} Y_{\square}, X_{\square} \text{g h i } Y_{\square} \rangle$$

then

$$\begin{aligned} \text{ih}(\delta, X, Y) &= \gamma_1 \cdot \text{LM}(\text{a b}) + \gamma_2 \cdot \text{LM}(\text{a b}) \\ &+ \gamma_1 \cdot \text{LM}(\text{e}) + \gamma_2 \cdot \text{LM}(\text{e f}) \\ &+ \gamma_1 \cdot \text{LM}(\text{g}) + \gamma_2 \cdot \text{LM}(\text{g h}) \\ &+ \gamma_3 \cdot \text{LM}(\text{g h i}) \end{aligned}$$

The coefficients of the combination are free parameters that we can tune to trade off between more pruning and more admissability. Setting the coefficients to zero gives perfect admissability but is also weak.

The heuristic for the first precondition node is computed similarly:

$$\begin{aligned} h(\delta, X, *) &= \text{cost}(\text{rule}(X, *, 1)) \\ &+ \text{cost}(\text{item}(\alpha, \delta, X, 1)) \\ &+ \text{cost}(\text{item}(\delta, \beta, *, 1)) \\ &+ \text{ih}(\delta, X, *) \end{aligned}$$

Standard CP			Augmented CP		
nodes (k)	BLEU	time	nodes (k)	BLEU	time
80	34.9	2.5	52	34.7	1.9
148	36.1	3.9	92	35.9	2.4
345	37.2	7.9	200	37.3	5.4
520	37.7	13.4	302	37.7	8.5
725	38.2	17.1	407	38.0	10.7
1092	38.3	27.1	619	38.2	16.3
1812	38.6	45.9	1064	38.5	27.7

Table 1: Results of standard and augmented cube pruning. The number of (thousands of) search nodes visited is given along with BLEU and average time to decode one sentence, in seconds.

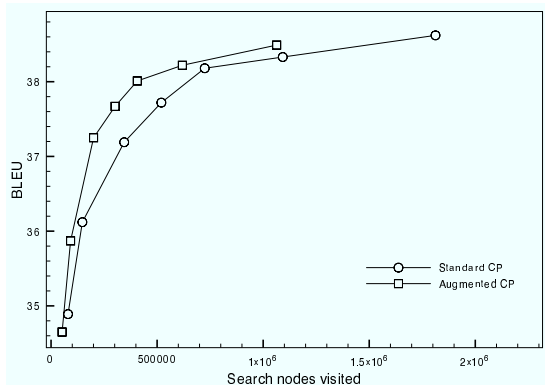


Figure 6: Nodes visited by standard and augmented cube pruning.

We also apply analogous heuristics to the subspan nodes.

5.2 Experimental setup

We evaluated all of the algorithms in this paper on a syntax-based Arabic-English translation system based on (Galley et al., 2006), with rules extracted from 200 million words of parallel data from NIST 2008 and GALE data collections, and with a 4-gram language model trained on 1 billion words of monolingual English data from the LDC Gigaword corpus. We evaluated the system’s performance on the NIST 2008 test corpus, which consists of 1357 Arabic sentences from a mixture of newswire and web domains, with four English reference translations. We report BLEU scores (Papineni et al., 2002) on untokenized, recapitalized output.

5.3 Results for Augmented Cube Pruning

The results for augmented cube pruning are compared against cube pruning in Table 1. The data

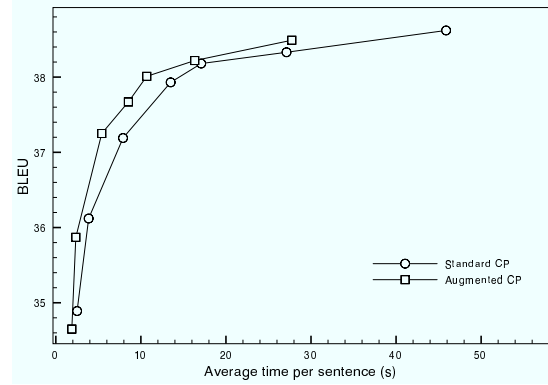


Figure 7: Time spent by standard and augmented cube pruning, average seconds per sentence.

	Standard CP	Augmented CP
subspan	12936	12792
precondition	851458	379954
rule	33734	33331
item	119703	118889
goal	74618	74159
TOTAL	1092449	619125
BLEU	38.33	38.22

Table 2: Breakdown of visited search nodes by type (for a fixed beam size).

from that table are also plotted in Figure 6 and Figure 7. Each line gives the number of nodes visited by the heuristic search, the average time to decode one sentence, and the BLEU of the output. The number of items kept by each span (the beam) is increased in each subsequent line of the table to indicate how the two algorithms differ at various beam sizes. This also gives a more complete picture of the speed/BLEU tradeoff offered by each algorithm. Because the two algorithms make the same sorts of lookahead computations with the same implementation, they can be most directly compared by examining the number of visited nodes. Augmenting cube pruning with admissible heuristics on the precondition nodes leads to a substantial decrease in visited nodes, by 35-44%. The reduction in nodes converges to a consistent 40% as the beam increases. The BLEU with augmented cube pruning drops by an average of 0.1 compared to standard cube pruning. This is due to the additional inadmissibility of the interaction heuristic.

To see in more detail how the heuristics affect the search, we give in Table 2 the number of nodes of each type visited by both variants for one beam

size. The precondition heuristic enables A^* to prune more than half the precondition nodes.

6 Exact Cube Pruning

Common wisdom is that the speed of cube pruning more than compensates for its inexactness (recall that this inexactness is due to the fact that it uses A^* search with inadmissible heuristics). Especially when we move into transducer-based MT, the search space becomes so large that brute-force item generation is much too slow to be practical. Still, within the heuristic search framework we may ask the question: is it possible to apply strictly admissible heuristics to the cube pruning search space, and in so doing, create a version of cube pruning that is both fast *and* exact, one that finds the n best items for each span and not just n good items? One might not expect such a technique to outperform cube pruning in practice, but for a given use case, it would give us a relatively fast way of assessing the BLEU drop incurred by the inexactness of cube pruning.

Recall again that the total cost of a goal node is given by Equation (6), which has four terms. It is easy enough to devise strong lower bounds for the first three of these terms by extending the reasoning of Section 5. Table 3 shows these heuristics. The major challenge is to devise an effective lower bound on the fourth term of the cost function, the interaction heuristic, which in our case is the incremental language model cost.

We take advantage of the following observations:

1. *In a given span, many boundary word patterns are repeated.* In other words, for a particular span $[\alpha, \beta]$ and carry κ , we often see many items of the form $[\alpha, \beta, X, \kappa]$, where the only difference is the postcondition X .
2. *Most rules do not introduce lexical items.* In other words, most of the grammar rules have the form $Z \rightarrow \langle X_0 Y_1, X_0 Y_1 \rangle$ (concatenation rules) or $Z \rightarrow \langle X_0 Y_1, Y_1 X_0 \rangle$ (inversion rules).

The idea is simple. We split the search into three searches: one for concatenation rules, one for inversion rules, and one for lexical rules. Each search finds the n -best items that can be created using its respective set of rules. We then take these $3n$ items and keep the best n .

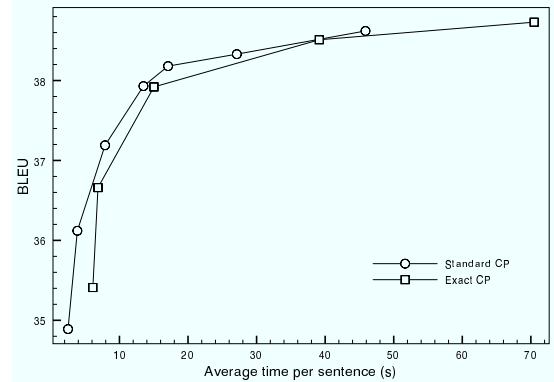


Figure 8: Time spent by standard and exact cube pruning, average seconds per sentence.

Doing this split enables us to precompute a strong and admissible heuristic on the interaction cost. Namely, for a given span $[\alpha, \beta]$, we precompute $ih_{adm}(\delta, X, Y)$, which is the best LM cost of combining carries from $items(\alpha, \delta, X)$ and $items(\delta, \beta, Y)$. Notice that this statistic is only straightforward to compute once we can assume that the rules are concatenation rules or inversion rules. For the lexical rules, we set $ih_{adm}(\delta, X, Y) = 0$, an admissible but weak heuristic that we can fortunately get away with because of the small number of lexical rules.

6.1 Results for Exact Cube Pruning

Computing the $ih_{adm}(\delta, X, Y)$ heuristic is not cheap. To be fair, we first compare exact CP to standard CP in terms of overall running time, including the computational cost of this overhead. We plot this comparison in Figure 8. Surprisingly, the time/quality tradeoff of exact CP is extremely similar to standard CP, suggesting that exact cube pruning is actually a practical alternative to standard CP, and not just of theoretical value. We found that the BLEU loss of standard cube pruning at moderate beam sizes was between 0.4 and 0.6.

Another surprise comes when we contrast the number of visited search nodes of exact CP and standard CP. See Figure 9. While we initially expected that exact CP must visit fewer nodes to make up for the computational overhead of its expensive heuristics, this did not turn out to be the case, suggesting that the computational cost of standard CP’s lookahead heuristics is just as expensive as the precomputation of $ih_{adm}(\delta, X, Y)$.

heuristic components						
	subspan $h(\delta)$	precondition1 $h(\delta, X)$	precondition2 $h(\delta, X, Y)$	rule $h(\delta, X, Y, i)$	item1 $h(\delta, X, Y, i, j)$	item2 $h(\delta, X, Y, i, j, k)$
r	rule(*, *, 1)	rule(X, *, 1)	rule(X, Y, 1)	rule(X, Y, i)	rule(X, Y, i)	rule(X, Y, i)
ι_1	item($\alpha, \delta, *, 1$)	item($\alpha, \delta, X, 1$)	item($\alpha, \delta, X, 1$)	item($\alpha, \delta, X, 1$)	item(α, δ, X, j)	item(α, δ, X, j)
ι_2	item($\delta, \beta, *, 1$)	item($\delta, \beta, *, 1$)	item($\delta, \beta, Y, 1$)	item($\delta, \beta, Y, 1$)	item($\delta, \beta, Y, 1$)	item(δ, β, Y, k)
ih	$ih_{adm}(\delta, *, *)$	$ih_{adm}(\delta, X, *)$	$ih_{adm}(\delta, X, Y)$	$ih_{adm}(\delta, X, Y)$	$ih_{adm}(\delta, X, Y)$	$ih_{adm}(\delta, X, Y)$

Table 3: Admissible heuristics for exact CP. We attach heuristic $h(\delta, X, Y, i, j, k)$ to the search node reached by choosing subspans $[\alpha, \delta]$, $[\delta, \beta]$, preconditions X and Y, the i^{th} rule of rules(X, Y), the j^{th} item of item(α, δ, X), and the k^{th} item of item(δ, β, Y). To form the heuristic for a particular type of search node (column), compute the following: $\text{cost}(r) + \text{cost}(\iota_1) + \text{cost}(\iota_2) + ih$

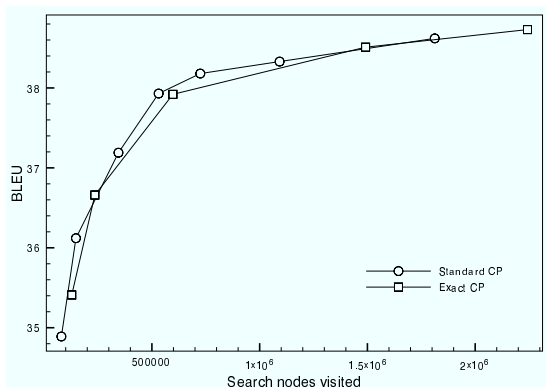


Figure 9: Nodes visited by standard and exact cube pruning.

7 Implications

This paper’s core idea is the utility of framing CKY item generation as a heuristic search problem. Once we recognize cube pruning as nothing more than A* on a particular search space with particular heuristics, this deeper understanding makes it easy to create faster and exact variants for other use cases (in this paper, we focus on tree-to-string transducer-based MT). Depending on one’s own particular use case, a variety of possibilities may present themselves:

1. *What if we try different heuristics?* In this paper, we do some preliminary inquiry into this question, but it should be clear that our minor changes are just the tip of the iceberg. One can easily imagine clever and creative heuristics that outperform the simple ones we have proposed here.
2. *What if we try a different search space?* Why are we using this particular search space? Perhaps a different one, one that makes de-

isions in a different order, would be more effective.

3. *What if we try a different search algorithm?* A* has nice guarantees (Dechter and Pearl, 1985), but it is space-consuming and it is not anytime. For a use case where we would like a finer-grained speed/quality tradeoff, it might be useful to consider an anytime search algorithm, like depth-first branch-and-bound (Zhang and Korf, 1995).

By working towards a deeper and unifying understanding of the smorgasbord of current MT speed-up techniques, our hope is to facilitate the task of implementing such methods, combining them effectively, and adapting them to new use cases.

Acknowledgments

We would like to thank Abdessamad Echihabi, Kevin Knight, Daniel Marcu, Dragos Munteanu, Ion Muslea, Radu Soricut, Wei Wang, and the anonymous reviewers for helpful comments and advice. Thanks also to David Chiang for the use of his LaTeX macros. This work was supported in part by CCS grant 2008-1245117-000.

References

- David Chiang. 2007. Hierarchical phrase-based translation. *Computational Linguistics*, 33(2):201–228.
- Rina Dechter and Judea Pearl. 1985. Generalized best-first search strategies and the optimality of a*. *Journal of the ACM*, 32(3):505–536.
- John DeNero, Mohit Bansal, Adam Pauls, and Dan Klein. 2009. Efficient parsing for transducer grammars. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*.

- Michel Galley, Mark Hopkins, Kevin Knight, and Daniel Marcu. 2004. What's in a translation rule? In *Proceedings of HLT/NAACL*.
- Michel Galley, Jonathan Graehl, Kevin Knight, Daniel Marcu, Steve DeNeefe, Wei Wang, and Ignacio Thayer. 2006. Scalable inference and training of context-rich syntactic models. In *Proceedings of ACL-COLING*.
- Liang Huang and David Chiang. 2007. Forest rescoring: Faster decoding with integrated language models. In *Proceedings of ACL*.
- Robert C. Moore and Chris Quirk. 2007. Faster beam-search decoding for phrasal statistical machine translation. In *Proceedings of MT Summit XI*.
- Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318.
- Judea Pearl. 1984. *Heuristics*. Addison-Wesley.
- Slav Petrov, Aria Haghghi, and Dan Klein. 2008. Coarse-to-fine syntactic machine translation using language projections. In *Proceedings of EMNLP*.
- Michael Pust and Kevin Knight. 2009. Faster mt decoding through pervasive laziness. In *Proceedings of NAACL*.
- Brian Roark and Kristy Hollingshead. 2008. Classifying chart cells for quadratic complexity context-free inference. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, pages 745–752.
- Brian Roark and Kristy Hollingshead. 2009. Linear complexity context-free parsing pipelines via chart constraints. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 647–655, Boulder, Colorado, June. Association for Computational Linguistics.
- Weixiong Zhang and Richard E. Korf. 1995. Performance of linear-space search algorithms. *Artificial Intelligence*, 79(2):241–292.
- Hao Zhang, Liang Huang, Daniel Gildea, and Kevin Knight. 2006. Synchronous binarization for machine translation. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 256–263.