

# Faster Cube Pruning

Andrea Gesmundo, James Henderson

Department of Computer Science  
University of Geneva

{andrea.gesmundo, james.henderson}@unige.ch

## Abstract

Cube Pruning is a fast method to explore the search space of a beam decoder. In this paper we present two modifications of the algorithm that aim to improve the speed and reduce the amount of memory needed at execution time. We show that, in applications where Cube Pruning is applied to a monotonic search space, the proposed algorithms retrieve the same  $K$ -best set with lower complexity. When tested on an application where the search space is approximately monotonic (Machine Translation with Language Model features), we show that the proposed algorithms obtain reductions in execution time with no change in performance.

## 1. Introduction

Since its first appearance in [2], Cube Pruning (CP) gained popularity in the Machine Translation community. Hierarchical or Tree-based Machine Translation models rely on Cube Pruning for decoding. Current research in such Tree-based models builds translation systems that more closely model the underlying recursive structure of languages, and produce more syntactically meaningful translations than Phrase-Based models. On the other hand, the non-syntactically motivated Phrase-Based Models have linear decoding time, because in practice they use a reordering limit. For Tree-based models, decoding is not linear with respect to sentence length. This higher complexity of Tree-based model decoding leads to a higher execution time. For this reason, real time systems that involve an automatic translation step (such as spoken dialogue systems, or real time text translation systems) often prefer Phrase Based models. With this work we investigate how to improve the speed of Tree-based models by reducing the Cube Pruning decoding complexity without affecting the accuracy of the decoding.

In [2], Cube Pruning is presented as an adaptation of the Lazy Algorithm [6] to the Hiero Model [1]. The name “Cube” Pruning comes from the fact that Hiero Model uses a binarised Synchronous Context Free Grammar; therefore at each node of the decoding hypergraph any combination of a rule with the two possible children must be considered, resulting in a three dimensional search space. But the use of Cube Pruning is not necessarily limited to the Machine Translation field or to a three dimensional search space. Lately CP is referred to as a general method for fast and

inexact beam decoding that can be applied whenever certain conditions occur. [5] shows that CP can be viewed as a specialisation of A\* for a specific search space with specific heuristics. CP’s generality allows its application to other tasks such as Parsing [6] and Word Alignment [8]. Any task using CP can take advantage of the improvements described in this paper, either in monotonic search spaces or in cases where there are perturbation factors that could create areas of non-monotonicity in the search space. This second case of an approximately monotonic search space occurs in many real word applications, such as the case of Machine Translation decoding with Language Model features, as clearly explained in Section 5 of [2].

We begin by describing the monotonic search space case in Section 2, first stating the problem formally, then reviewing the Lazy Algorithm and introducing the two algorithms we propose as improvement. Section 3 describes the approximately monotonic search space case. It explains how Cube Pruning handles the perturbations of the search space and then shows how the two proposed algorithms can be extended as well. Section 4 reports experiments to test the proposed algorithm and to compare them with Cube Pruning. Finally, Section 5 further investigates the differences in behaviour between Cube Pruning and the proposed algorithms.

## 2. Monotonic Search Space

Cube Pruning is a solution to the problem of finding the best  $k$  results of the product of  $n$  elements selected from each of  $n$  ordered lists. All the applications of Cube Pruning can be reduced to this simple problem. Formally we can state the problem as follows:

**Problem 1.** *As input we have:*

- A set  $\mathcal{L}$ , containing  $N = |\mathcal{L}|$  ordered lists  $L_n \mid 1 \leq n \leq N$ . We refer to the  $i$ -th element of the list  $L_n$  as  $x_{n,i} \in \mathcal{X}$ .
- An ordering function  $\min_{\leq}$ , that allows us to compare elements of  $\mathcal{X}$ , and sort the lists in  $\mathcal{L}$ .
- An operator  $\odot$  with monotonic property, so that if  $x', x'', \hat{x} \in \mathcal{X}$  and  $x' \min_{\leq} x''$  then  $(x' \odot \hat{x}) \min_{\leq} (x'' \odot \hat{x})$

- The size of the output beam  $k$ .

As solution of the problem we want:

- The ordered list “best- $k$ ” containing the top  $k$  elements of  $\mathcal{P}$ . Where  $\mathcal{P}$  is the set of the results obtained from the  $\odot$ -product of  $n$  elements selected from each of the  $n$  ordered lists in  $\mathcal{L}$ :

$$\mathcal{P} \equiv \left\{ \bigodot_{1 \leq n \leq N} x_{n,i_n} \mid x_{n,i_n} \in L_n, 1 \leq i_n \leq |L_n| \right\} \quad (1)$$

Formally:

$$\text{best-}k \equiv \{x_i \mid 1 \leq i \leq k, x_i \in \mathcal{P}, \forall \hat{x} \in (\mathcal{P} - \text{best-}k), \hat{x} \min_{\leq} x_i\} \quad (2)$$

Following [4], it is possible to interpret  $\odot$  and  $\min_{\leq}$  respectively as  $\otimes$  and  $\oplus$  (the two binary operations of a semiring) and show the relation between the stated problem and the general definition of bottom-up parsing.

We show the correspondence between Problem 1 and common problems where Cube Pruning is applied. As an example, consider a Probabilistic Context Free Grammar (PCFG) Parser. Problem 1 is the one that must be solved during bottom up decoding at each node of the hypergraph. The PCFG decoder has to choose the best  $k$  derivations that lead to the generation of the span  $[i, j]$  associated with the node. If using only context free features, the score of the derivation is given by summing the log-probability score of the rule  $R$  with the scores of the derivations related to the children of  $R$ .

To show the correspondence between Problem 1 and the parsing example, given a rule  $R$ , we make these associations:

- $x_{n,i}$  is the score of a derivation for the  $n$ -th child of  $R$ .
- $L_n$  is the ordered list of scores of the derivations correspondent to the  $n$ -th child of  $R$ .
- $\mathcal{X}$  is the set of possible values for the scores, for example  $\mathbb{R}$ , set of the real numbers.
- The two binary functions  $\min_{\leq}$  and  $\odot$  are respectively  $\leq$  and  $+$ .

The same correspondence holds for hierarchical Machine Translation decoding, which can be seen as a Synchronous PCFG decoder. Similarly we can find correspondences with other problems whose solution involves the use of Cube Pruning, since they are based on bottom-up parsing techniques, as for example in the case of [8] for Hierarchical Word Alignment.

## 2.1. Lazy Algorithm

We describe here how to solve the stated problem using the Lazy Algorithm presented in [6]. This method is the base for the Cube Pruning algorithm. CP extends the Lazy Algorithm to approximately-monotonic search spaces, as discussed in section 3.

If the lists in  $\mathcal{L}$  were not sorted, a brute force solution would be to compute all the possible results, sort them and pick the best  $k$ . But having defined an ordering function  $\min_{\leq}$  and a monotonic operator  $\odot$ , it is possible to find the solution without computing all the possible results.

To simplify the explanation, we introduce new notation. We denote  $\hat{x} = x_{1,i_1} \odot x_{2,i_2} \odot \dots \odot x_{N,i_N}$  with  $\hat{x} = \phi(\mathbf{v})$ , where  $\mathbf{v}$  is the vector of the indexes of the elements in the lists  $L_n$ ,  $\mathbf{v} \equiv \langle i_1, i_2, \dots, i_N \rangle$ . We can interpret  $\mathbf{v}$  as coordinates in a  $N$ -dimensional space, that is our search space  $\mathcal{S}$ , and  $\phi(\cdot)$  can be seen as a function that maps points in the search space  $\mathcal{S}$  to values in  $\mathcal{X}$ :  $\mathcal{P} \equiv \{\phi(\mathbf{u}) \mid \mathbf{u} \in \mathcal{S}\}$ . We define  $\mathbf{1}$  to be the vector of length  $N$  whose elements are all 1, and define  $\mathbf{b}^i$  to be the vector of length  $N$  whose elements are all 0 except for  $b^i_i = 1$ .

The Lazy Algorithm solves Problem 1 proceeding recursively. At each iteration a new element of the best- $k$  list is found. The element  $\phi(\mathbf{1})$  is added to the best- $k$  list at the first iteration. Because of our assumptions, we know that  $\phi(\mathbf{1})$  is the best element in  $\mathcal{S}$ . After adding an element to the best- $k$  list, all the neighbouring elements are visited. Formally the set of neighbouring elements  $\mathcal{V}_{\mathbf{v}}$  of  $\mathbf{v}$  is defined as:

$$\mathcal{V}_{\mathbf{v}} \equiv \{\mathbf{u} = \mathbf{v} + \mathbf{b}^i \mid 1 \leq i \leq N, \mathbf{u} \in \mathcal{S}\} \quad (3)$$

Then each neighbour is added to a Queue of Candidates  $Q$  if it is not already in it. For all the iterations following the first, the best element of the queue is moved to the best- $k$  and the loop iterate inserting in  $Q$  all its neighbours. The algorithm terminates when either the Queue is empty or the best- $k$  list has  $k$  elements. So the number of iterations is  $\min(|\mathcal{S}|, k)$ . Figure 1 (a) pictures an example for the Lazy Algorithm for a bi-dimensional case  $N = 2$ .

Algorithm 1 illustrates the Lazy Algorithm. At *line 2* the queue  $Q$  is initialised inserting the origin element  $\mathbf{1}$ . At *line 3* the best- $k$  list is initialised as an empty list. The main loop starts at *line 4*, that will terminate either when the best- $k$  list is full, or when the queue is empty; this second case occurs when the search space contains less than  $k$  elements. At *line 5*, as first step of the loop, the best element in  $Q$  is popped and named  $\mathbf{v}$ . At *line 6*  $\mathbf{v}$  is inserted in the best- $k$ . From *line 7* to *line 11* the algorithm iterates over all the neighbouring elements of  $\mathbf{v}$ , and add them to  $Q$ . The test at *line 8* checks that the element to be inserted in  $Q$  is not already into  $Q$ . The pseudocode for the function that retrieves the neighbouring elements is listed between *line 13* and *line 20*. This function implements the definition of  $\mathcal{V}_{\mathbf{v}}$ . At *line 14*  $\mathcal{V}_{\mathbf{v}}$  is initialised as an empty set. The loop between *line 15* and *line 20* iterates over all the coordinates of  $\mathbf{v}$ . Each neighbour is created at *line 16* adding 1 to  $\mathbf{v}$  at the current coordinate. The new element is added in  $\mathcal{V}_{\mathbf{v}}$  at *line 18*. The test at *line 17* is needed to check that the incremented coordinate is still inside the valid range of values, and points to an actual element in  $L_i$ .

To find the best  $k$  elements, the Lazy Algorithm avoids computing the  $\phi(\cdot)$  score for all the elements in the search

---

**Algorithm 1** Lazy Algorithm
 

---

```

1: function MainLoop ( $\mathcal{L}$ ,  $\odot$ ,  $\min_{\leq}$ ,  $k$ ): best- $k$ 
2:  $Q \leftarrow \{1\}$ ;
3: best- $k \leftarrow$  empty list;
4: while  $|\text{best-}k| < k$  and  $|Q| > 0$  do
5:    $\mathbf{v} \leftarrow \text{get-}\min_{\leq}(Q)$ ;
6:   insert(best- $k$ ,  $\mathbf{v}$ );
7:   for all  $\mathbf{v}' \in \text{NeighboursOf}(\mathbf{v})$  do
8:     if  $\mathbf{v}' \notin Q$  then
9:       insert( $Q$ ,  $\mathbf{v}'$ );
10:    end if
11:  end for
12: end while

13: procedure NeighboursOf ( $\mathbf{v}$ ):  $\mathcal{V}_{\mathbf{v}}$ 
14:  $\mathcal{V}_{\mathbf{v}} \leftarrow$  empty set;
15: for  $1 \leq i \leq |\mathbf{v}|$  do
16:    $\mathbf{v}^* \leftarrow \mathbf{v} + \mathbf{b}^i$ 
17:   if  $v_i^* \leq |L_i|$  then
18:     insert( $\mathcal{V}_{\mathbf{v}}$ ,  $\mathbf{v}^*$ );
19:   end if
20: end for

```

---

space  $\mathcal{S}$ . It just needs to compute those best- $k$  along with few other elements that remain in the candidate queue. Note that in the monotonic search space there is no approximation; the algorithm finds the correct best- $k$  set. This will not be true for the approximately monotonic case, as we will see in Section 3.

To improve the Lazy Algorithm we cannot avoid computing the best  $k$  elements. What can be done is to reduce the number of elements that are in the queue during execution time, and the number of leftover elements that remain in the queue at the end of the execution. A shorter queue needs less memory at run time and consumes less time adding the new elements in sorted order. Furthermore, if the queue at the end of the execution is shorter, we save computation time avoiding computing elements that are not relevant for the output.

## 2.2. Algorithm 2

Algorithm 2 is the first of the two algorithms we propose as improvement. To introduce it, we focus on a detail of the Lazy Algorithm: before adding a neighbour element in  $Q$ , the algorithm needs to check that it has not already been added (see Algorithm 1 *line* 8). At the implementation level, this check requires either a search in  $Q$  ( $\mathcal{O}(\log|Q|)$ ) or an additional hash table that allows finding the response in constant time but needs to be maintained as  $Q$  changes. This check is needed because an element can be in the set of neighbours of more than one parent element.

To define the new algorithm we change the definition of neighbouring elements  $\mathcal{V}_{\mathbf{v}}$ . For an element  $\mathbf{v}$  we denote the set of parent elements with

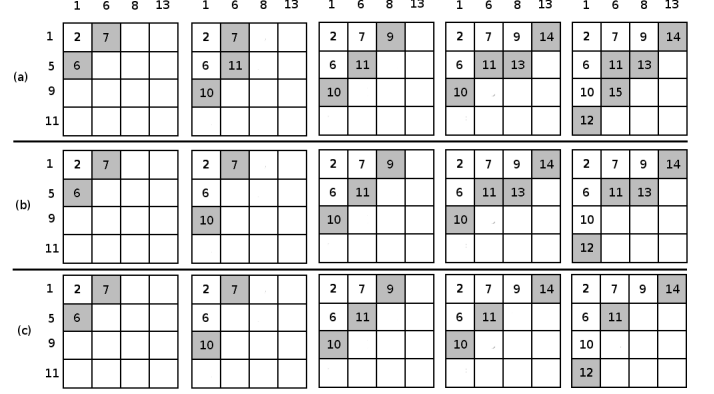


Figure 1: Examples of processing for the three algorithms. As input is given  $\mathcal{L}$  with  $N = 2$ , and  $L_1 \equiv \{1, 5, 9, 11\}$ ,  $L_2 \equiv \{1, 6, 8, 13\}$  sorted lists. For each algorithm are depicted the first 5 iterations. Shaded cells denote elements in the candidate queue, white cells denote elements added in the best- $k$  output list. (a) Lazy Algorithm, (b) Algorithm 2 having vertical dimension with index 1 and horizontal dimension with index 2, (c) Algorithm 3.

$$\mathcal{A}_{\mathbf{v}} \equiv \{\mathbf{u} | \mathbf{v} \in \mathcal{V}_{\mathbf{u}}, \mathbf{u} \in \mathcal{S}\} \quad (4)$$

We define a new neighbouring elements set  $\mathcal{V}'_{\mathbf{v}}$  such that  $\mathcal{A}'_{\mathbf{v}}$  contains a single element. We propose the following definition for  $\mathcal{V}'_{\mathbf{v}}$ :

$$\mathcal{V}'_{\mathbf{v}} \equiv \{\mathbf{u} = \mathbf{v} + \mathbf{b}^i | \forall j < i, v_j = 1, 1 \leq i \leq N, \mathbf{u} \in \mathcal{S}\} \quad (5)$$

Where  $v_j$  is the  $j$ -th coordinate of the vector  $\mathbf{v} = \langle v_1, v_2, \dots, v_N \rangle$ . As we can see from the definition,  $\mathcal{V}'_{\mathbf{v}} \subseteq \mathcal{V}_{\mathbf{v}}$ .  $\mathcal{V}'_{\mathbf{v}}$  contains the elements of  $\mathcal{V}_{\mathbf{v}}$  that have all unit values for the coordinates preceding the one updated. For example:  $\mathbf{v} + \mathbf{b}^1$  is always added in  $\mathcal{V}'_{\mathbf{v}}$ ;  $\mathbf{v} + \mathbf{b}^2$  is added only if  $v_1 = 1$ ; and in general  $\mathbf{v} + \mathbf{b}^i$  is added only if  $\{v_1, v_2, \dots, v_{i-1}\}$  are all 1. In this setting, it becomes necessary to have an order of the coordinates of  $\mathbf{v}$ . This means that we need to set an ordering of the elements in  $\mathcal{L}$ , and  $\mathcal{L}$  becomes an ordered list rather than a set. This can be seen as deciding an order for the dimensions of the search space  $\mathcal{S}$ .

Having defined  $\mathcal{V}'_{\mathbf{v}}$ , we can consider the set of parent elements of  $\mathbf{v}$ :  $\mathcal{A}'_{\mathbf{v}} \equiv \{\mathbf{u} | \mathbf{v} \in \mathcal{V}'_{\mathbf{u}}, \mathbf{u} \in \mathcal{S}\}$ . Now we prove that  $\mathcal{A}'_{\mathbf{v}}$  contains only one element for any  $\mathbf{v} \neq \mathbf{1}$ .

**Theorem 1.** For any element  $\mathbf{v} \neq \mathbf{1}$  in the search space  $\mathcal{S}$ , we have that  $|\mathcal{A}'_{\mathbf{v}}| = 1$ .

*Proof.* By contradiction:

- Assume that there is an element  $\mathbf{v}$  for which  $|\mathcal{A}'_{\mathbf{v}}| > 1$ .
- Let  $\mathbf{u}, \mathbf{w} \in \mathcal{A}'_{\mathbf{v}}, \mathbf{u} \neq \mathbf{w}$ .

- From the definition of  $\mathcal{A}'_{\mathbf{v}}$ , we have that:  $\mathbf{v} \in \mathcal{V}'_{\mathbf{u}}$  and  $\mathbf{v} \in \mathcal{V}'_{\mathbf{w}}$
- From the definition of  $\mathcal{V}'$ , we have that:  $\mathbf{v} = \mathbf{u} + \mathbf{b}^i$ ,  $\mathbf{v} = \mathbf{w} + \mathbf{b}^j$  and we have also that  $u_x = 1 \forall x < i$ ,  $w_y = 1 \forall y < j$ , with  $1 \leq i, j \leq N$ .
- $\mathbf{v} = \mathbf{u} + \mathbf{b}^i \wedge \mathbf{v} = \mathbf{w} + \mathbf{b}^j \Rightarrow \mathbf{u} = \mathbf{w} + \mathbf{b}^j - \mathbf{b}^i$

To conclude we distinguish three cases:

- if  $i = j$  then:  
 $\mathbf{u} = \mathbf{w} + \mathbf{b}^j - \mathbf{b}^i = \mathbf{w}$  and this cannot be because it is in contradiction with  $\mathbf{u} \neq \mathbf{w}$ .
- if  $i < j$  then:  
From definition of  $\mathbf{b}^i$  we have that:  $b_i^j = 0, b_i^i = 1$   
From  $i < j$  and the definition of  $\mathcal{V}'$  we have that:  $w_i = 1$   
Therefore:  $u_i = w_i + b_i^j - b_i^i = 0$   
But this assignment of values cannot be because no vector  $\mathbf{v} \in \mathcal{S}$  can have coordinate with value 0 since the indexing of the lists in  $\mathcal{L}$  start from 1 by definition.
- Similarly if  $i > j$ .

We showed by contradiction that  $\mathcal{A}'_{\mathbf{v}}$  cannot contain two different elements. Therefore  $|\mathcal{A}'_{\mathbf{v}}| \leq 1$ .

Now consider any  $\mathbf{v} \neq \mathbf{1}$ , and let  $i$  be the index of the first coordinate in the sequence so that  $v_i > 1$ . We compute  $\mathbf{u} = \mathbf{v} - \mathbf{b}^i$ , then from the definition of  $\mathcal{V}'$  we have that  $\mathbf{v} \in \mathcal{V}'_{\mathbf{u}}$ . And we can state that for  $\mathbf{v} \neq \mathbf{1}$  we have that  $|\mathcal{A}'_{\mathbf{v}}| > 0$ .

Combining  $|\mathcal{A}'_{\mathbf{v}}| \leq 1$  and  $|\mathcal{A}'_{\mathbf{v}}| > 0$ , it follows that  $|\mathcal{A}'_{\mathbf{v}}| = 1$ .  $\square$

Having defined how to retrieve the neighbours, it's straightforward to write the new algorithm, given as pseudocode in Algorithm 2.

Algorithm 2 does not need to check if the neighbours are already in  $Q$ . Therefore the test that was done at *line* 8 of Algorithm 1 is removed. This allows us to avoid the use of a hash table or the execution of a  $\mathcal{O}(\log|Q|)$  search on  $Q$ . The new *NeighboursOf()* procedure is really similar to the first version. Only the test at *line*18 is added to break the loop when the first coordinate in the sequence that has a value different from 1 is found. The loop break leads to a smaller set of neighbours and therefore a smaller  $Q$  and less time spent in the loop. Since most of the elements in the search space have first coordinate with value different from 1, the loop will execute only the first round in most of the cases. Figure 1 (b) illustrates an example for Algorithm 2 for a bi-dimensional case  $N = 2$ , where the vertical dimension has index 1 and the horizontal has index 2.

Having defined this new algorithm we need to prove that it solves Problem 1. The proof below shows that the Algorithm 2 outputs the best- $k$  list, that contains the top  $k$  elements in  $\mathcal{S}$ .

---

### Algorithm 2

---

```

1: function MainLoop ( $\mathcal{L}, \odot, \min_{\leq}, k$ ): best- $k$ 
2:  $Q \leftarrow \{\mathbf{1}\}$ ;
3: best- $k \leftarrow$  empty list;
4: while  $|\text{best-}k| < k$  and  $|Q| > 0$  do
5:    $\mathbf{v} \leftarrow \text{get-}\min_{\leq}(Q)$ ;
6:   insert(best- $k, \mathbf{v}$ );
7:   for all  $\mathbf{v}' \in \text{NeighboursOf}(\mathbf{v})$  do
8:     insert( $Q, \mathbf{v}'$ );
9:   end for
10: end while

11: procedure NeighboursOf ( $\mathbf{v}$ ):  $\mathcal{V}'_{\mathbf{v}}$ 
12:  $\mathcal{V}'_{\mathbf{v}} \leftarrow$  empty set;
13: for  $1 \leq i \leq |\mathbf{v}|$  do
14:    $\mathbf{v}^* \leftarrow \mathbf{v} + \mathbf{b}^i$ 
15:   if  $v_i^* \leq |L_i|$  then
16:     insert( $\mathcal{V}'_{\mathbf{v}}, \mathbf{v}^*$ );
17:   end if
18:   if  $v_i \neq 1$  then
19:     break;
20:   end if
21: end for

```

---

**Theorem 2.** *Algorithm 2 solves Problem 1.*

*Proof.* By induction on the iterations:

- Base case:

At the first iteration,  $\mathbf{1}$  is added to best- $k$ . Having monotonic search space and ordered lists, this is the best element in  $\mathcal{S}$ .

- Induction step:

At the  $x$ -th iteration, the best- $k$  list contains already the top  $x - 1$  elements and a new element  $\mathbf{v}$  is added from  $Q$ . Now we focus on the the set of elements in the search space that are “better” than  $\mathbf{v}$ , formally:  $\mathcal{B} \equiv \{\mathbf{u} | \phi(\mathbf{u}) \leq \phi(\mathbf{v}), \mathbf{u} \in \mathcal{S}\}$ . Then we distinguish three cases:

- $\mathbf{u} \in \mathcal{B}$  is a candidate in  $Q$ : This event cannot occur because in this case  $\mathbf{u}$  would have been chosen instead of  $\mathbf{v}$ .
- $\mathbf{u} \in \mathcal{B}$  has not been considered ( $\mathbf{u} \notin Q, \mathbf{u} \notin \text{best-}k$ ): Now we recursively apply  $\mathcal{A}'$  to  $\mathbf{u}$  and its ancestors, until we find the unique path that connects  $\mathbf{1}$  with  $\mathbf{u}$ , *path*:  $\langle \mathbf{1}, \mathbf{w}_1, \dots, \mathbf{w}_l, \mathbf{u} \rangle$ .

From the monotonicity of search space and definition of  $\mathcal{A}'$  we have that:  $\phi(\mathbf{1}) \leq \phi(\mathbf{w}_1) \leq \dots \leq \phi(\mathbf{w}_l) \leq \phi(\mathbf{u})$ . Therefore all elements in *path* are in  $\mathcal{B}$ . Knowing that  $\mathbf{1}$  is already in best- $k$  and  $\mathbf{u}$  hasn't been added in  $Q$ , from the definition of Algorithm 2, we know that there is an element in *path* that

has been added in  $Q$  but has not yet been moved into best- $k$ . But this cannot happen since there would be a  $\mathbf{w} \in \mathcal{B}$  candidate in  $Q$ .

- $\mathbf{u} \in \mathcal{B}$  is in best- $k$ : By exclusion this is the only possible case.

So we can state that all elements in  $\mathcal{B}$  are in best- $k$ :  $\mathcal{B} \subseteq \text{best-}k$ . Considering the loop invariant stated above: “at the  $x$ -th iteration, best- $k$  list contains already the top  $x - 1$  elements”, we know that best- $k$  cannot contain elements that are “worse” than  $\mathbf{v}$ , therefore:  $\mathcal{B} \equiv \text{best-}k$ . We conclude that all the elements that are “better” than  $\mathbf{v}$  are already in best- $k$ , and there are no “better” element outside best- $k$  list. This proves that at the end of the  $x$ -th iteration, best- $k$  contains the best  $x$  elements.  $\square$

As we explained, for this algorithm we need to specify an order of the coordinates of the search space. It is indifferent in which order we put the coordinates, in the monotonic search space the algorithm will retrieve the same best- $k$ . But the set of elements that are in the queue will be different. For example, if in Figure 1 (b) we wanted to picture the same example but with reversed dimension order (horizontal dimension with index 1, and vertical with index 2), at the second step we would have the element with score 11 in  $Q$ , in the fourth step we would not have the element with score 13, and in the last step we would have the element with score 15 instead of the one with score 13. These differences in  $Q$  will affect the decoding in the approximately monotonic search space, as will be discussed in Section 3.

### 2.3. Algorithm 3

Now we introduce the second proposed variation of the Lazy Algorithm. To explain the intuition behind Algorithm 3, consider Figure 1 (a). At the first round we insert 2 in the best- $k$ , then 6 and 7 are added in the queue as candidates. In the second round we pick the smaller element from  $Q$  and move it to best- $k$ . Following Lazy Algorithm we need to add all the neighbours of 6 in the queue. Among the neighbours of 6 there is 11. Before adding 11 in  $Q$ , we could notice that one of its ancestors, 7, is still in  $Q$ . Thus it is certain that at the next round 11 will not be selected to be moved in best- $k$ . So its presence in  $Q$  is useless until 7 is selected. Anyway, after 7 is picked, the algorithm will try again to insert 11 in  $Q$ . Therefore we can avoid adding 11 at this step and just add 10 as neighbour of 6.

In general we can say that it is possible to avoid adding an element to  $Q$  until all its ancestors are in the best- $k$  list. Algorithm 3 uses the same  $\mathcal{V}_{\mathbf{v}}$  and  $\mathcal{A}_{\mathbf{v}}$  as the one defined for the Lazy Algorithm. The difference is in the policy with which  $Q$  is updated.

The pseudocode for Algorithm 3 is reported below. We can see that the pseudocode for Algorithm 3 is very similar to the pseudocode for the Lazy Algorithm. The only difference

---

### Algorithm 3

---

```

1: function MainLoop ( $\mathcal{L}, \odot, \text{min}_{\leq}, k$ ): best- $k$ 
2:  $Q \leftarrow \{1\}$ ;
3: best- $k \leftarrow$  empty list;
4: while |best- $k$ | <  $k$  and | $Q$ | > 0 do
5:    $\mathbf{v} \leftarrow \text{get-min}_{\leq}(Q)$ ;
6:   insert(best- $k, \mathbf{v}$ );
7:   for all  $\mathbf{v}' \in \text{NeighboursOf}(\mathbf{v})$  do
8:     if  $\mathcal{A}_{\mathbf{v}'} \subseteq \text{best-}k$  then
9:       insert( $Q, \mathbf{v}'$ );
10:    end if
11:  end for
12: end while

13: procedure NeighboursOf ( $\mathbf{v}$ ):  $\mathcal{V}_{\mathbf{v}}$ 
14:  $\mathcal{V}_{\mathbf{v}} \leftarrow$  empty set;
15: for  $1 \leq i \leq |\mathbf{v}|$  do
16:    $\mathbf{v}^* \leftarrow \mathbf{v} + \mathbf{b}^i$ 
17:   if  $\mathbf{v}_i^* \leq |L_i|$  then
18:     insert( $\mathcal{V}_{\mathbf{v}}, \mathbf{v}^*$ );
19:   end if
20: end for

```

---

is at *line 8*: before adding any of the neighbours of  $\mathbf{v}$ , we check that all the predecessors of the neighbour are already in the best- $k$  list:  $\mathcal{A}_{\mathbf{v}'} \subseteq \text{best-}k$ . Obviously we no longer need to check if the neighbour  $\mathbf{v}'$  is in the queue, because there is no risk that an element is added twice, since every element is added to the queue only in the iteration where the last of its predecessor is added in best- $k$ .

As we observed in Algorithm 2: avoiding the statement at *line 8* of the Lazy Algorithm allows us to avoid using an additional hash table. Unfortunately, for Algorithm 3 we replace that statement with:  $\mathcal{A}_{\mathbf{v}'} \subseteq \text{best-}K$ , that needs a similar data structure. Furthermore we need to repeat the query a number of times proportional to  $|\mathcal{A}_{\mathbf{v}'}|$ . Fortunately this number is proportional to the number of dimensions in the search space and that is constant or bounded in most cases. For example in Machine Translation it is preferable to use a grammar composed by binarised rules, in which case  $|\mathcal{A}_{\mathbf{v}'}| \leq 2$ .

Now we show that Algorithm 3 solves Problem 1. Since the proof is similar to the proof of Theorem 2, we just prove the step that is different. The rest of the proof is done by induction on the iterations exactly following the proof for Theorem 2.

**Theorem 3.** *At the  $x$ -th iteration of Algorithm 3, the best- $k$  list contains the top  $x - 1$  elements and a new element  $\mathbf{v}$  is added from  $Q$ . Let  $\mathcal{B} \equiv \{\mathbf{u} | \phi(\mathbf{u}) \leq \phi(\mathbf{v}), \mathbf{u} \in \mathcal{S}\}$ , being the set of elements in  $\mathcal{S}$  that are “better” than  $\mathbf{v}$ . Then  $\mathcal{B} \subseteq \text{best-}k$ .*

*Proof.*

- $\mathbf{u} \in \mathcal{B}$  is candidate in  $Q$ : In this case  $\mathbf{u}$  would have been picked instead of  $\mathbf{v}$ .

- $\mathbf{u} \in \mathcal{B}$  has not been considered ( $\mathbf{u} \notin Q, \mathbf{u} \notin \text{best-}k$ ): Let's define function  $\gamma(\cdot) : \mathcal{S} \rightarrow \mathbb{N}$ , so that  $\gamma(\mathbf{u}) = \sum_{i=1}^N u_i$ . From the monotonicity of the search space, we can state that:  $\mathcal{A}_{\mathbf{u}} \subseteq \mathcal{B}$ . Therefore  $\mathcal{A}_{\mathbf{u}} \cap Q = \emptyset$ . From the definition of the update policy of  $Q$  for Algorithm 3, we can say that not all the elements of  $\mathcal{A}_{\mathbf{u}}$  are in  $\text{best-}k$ , since otherwise  $\mathbf{u}$  would have already been added in  $Q$ . We conclude that there must be an element  $\mathbf{w}_1 \in \mathcal{A}_{\mathbf{u}}$  that has not been considered yet. If we apply the same reasoning recursively, we have that there exists a  $\mathbf{w}_2 \in \mathcal{A}_{\mathbf{w}_1}$  that has not been considered, and a  $\mathbf{w}_3 \in \mathcal{A}_{\mathbf{w}_2}$  that has not been considered, and so on. Now we can find where this sequence leads using the  $\gamma(\cdot)$  function: if  $\gamma(\mathbf{u}) = K$  then, from how  $\mathcal{V}_{\mathbf{w}_1}$  is defined, we have that  $\gamma(\mathbf{w}_1) = K - 1$  and  $\gamma(\mathbf{w}_2) = K - 2$  and so on. Repeating the iteration  $K - N$  times, we reach the element  $\mathbf{w}_x$  such that  $\gamma(\mathbf{w}_x) = N$ . Given that  $N = |\mathcal{L}|$  is the number of dimensions of the search space, the only element  $\mathbf{w}_x$  for which  $\gamma(\mathbf{w}_x) = N$  is  $\mathbf{1}$ . Now we have reached an absurd because  $\mathbf{1}$  cannot be in the chain of elements that have not been considered.

Having shown that every element in  $\mathcal{B}$  must have already been considered and cannot be in  $Q$ , we have to conclude that: all elements in  $\mathcal{B}$  have been already moved into  $\text{best-}k$ . Therefore  $\mathcal{B} \subseteq \text{best-}k$ .  $\square$

Compared to Lazy Algorithm, we expect Algorithm 3 to be faster since it uses the minimal number of candidates in the queue. Compared to Algorithm 2, Algorithm 3 also puts fewer elements in  $Q$  but needs to use an additional data structure, such as a hash table. Given this analysis, we can't be sure whether Algorithm 2 or Algorithm 3 is faster, so we will address this issue in the empirical experiments in section 4.

We have shown that the three algorithms return the same  $\text{best-}k$  list in a monotonic search space environment. In the next section we discuss the case of approximately monotonic search spaces, and how we expect the behaviour of the 3 algorithms to change. Later we will compare our expectations with the empirical results.

### 3. Approximately Monotonic Search Space

This section describes the approximately monotonic search space case, shows the correspondence with some popular applications, and describes how the algorithms introduced for the monotonic case can be extended.

In the approximately monotonic search space we still have the set of ordered lists  $\mathcal{L}$ , the ordering function  $\text{min}_{\leq}$ , and the monotonic operator  $\odot$  as we defined them in Section 2. The main difference is that instead of computing the  $\odot$ -product of the single input items:  $\hat{x} = x_{1,i_1} \odot x_{2,i_2} \odot \dots \odot x_{N,i_N}$ , in the approximately monotonic case we have to add a perturbation element:

$$\hat{x} = x_{1,i_1} \odot \dots \odot x_{N,i_N} \odot \Delta(i_1, \dots, i_N) \quad (6)$$

Where  $\Delta(i_1, \dots, i_N)$  is the perturbation function,  $\Delta(\cdot) : \mathcal{S} \rightarrow \mathcal{X}$ . For brevity we define  $\phi(\mathbf{v}) = \phi(\mathbf{v}) \odot \Delta(\mathbf{v})$ . The perturbation function  $\Delta(\cdot)$  must have two properties:

- $\Delta(\cdot)$  is not proportional to  $\phi(\cdot)$ , formally:  $\exists \mathbf{v}, \mathbf{u} \in \mathcal{S}, \phi(\mathbf{v}) \leq \phi(\mathbf{u})$  such that  $\Delta(\mathbf{v}) \not\leq \Delta(\mathbf{u})$ .

Therefore we could have cases where  $\phi(\mathbf{v}) \leq \phi(\mathbf{u})$ , but  $\psi(\mathbf{v}) \not\leq \psi(\mathbf{u})$ . In those cases the search space described by  $\psi(\cdot)$  has some perturbations and it could have areas of non-monotonicity.

- The magnitude of  $\Delta(\mathbf{v})$  must be smaller or proportional to  $\phi(\mathbf{v})$ . In cases where  $|\Delta(\mathbf{v})| \gg |\phi(\mathbf{v})|$  the search space could become fully non-monotonic. In order to successfully apply Cube Pruning and the proposed algorithms, we need to have an approximately monotonic search space.

Examples of adding a perturbation function  $\Delta(\cdot)$  are adding the Language Model in a Machine Translation system, or adding global features in a CKY parser. More generally, a perturbation function  $\Delta(\cdot)$  can add non-local contextual factors in the scores used to define Problem 1.

Given this definition of the approximately monotonic search space in terms of function  $\psi(\cdot)$ , to guarantee finding the  $\text{best-}k$  list would require exploring the entire space. This is because the function  $\Delta(\cdot)$  could be any function with the two stated properties, and any distortion in the search space could be introduced. If we apply the Lazy Algorithm to the approximately monotonic search space, it would compute a small corner of the space, and we would obtain an approximate  $\text{best-}k$  list:  $\tilde{\text{best-}k}$ . As described in [2], Cube Pruning applies the Lazy Algorithm using a margin parameter  $\epsilon$  to control the level of uncertainty about whether the true  $\text{best-}k$  has been found. Cube Pruning continues to search the space until the  $\tilde{\text{best-}k}$  contains  $k$  elements and all the elements  $\mathbf{v} \in \mathcal{S}$  such that  $\psi(\mathbf{v}) \leq \psi(\mathbf{u}^*) \odot \epsilon$  have been computed, where  $\mathbf{u}^*$  is the "worst" element in  $\tilde{\text{best-}k}$ . Thus the algorithm does not stop as soon as the list is full, but continues to search in the neighbourhood of the top corner of the search space for a while. Applications with a small ratio  $R = |\Delta(\mathbf{v})|/|\phi(\mathbf{v})|$  are closer to the monotonic case, and we can use a small  $\epsilon$  and still have good quality results. When  $R$  is big, we need to tune  $\epsilon$  to a higher value, which will slow down execution because more of the space is searched. In either case, Cube Pruning is not an exact search technique and will return an approximate  $\tilde{\text{best-}k}$  list.

We can apply the same reasoning that led us from Lazy Algorithm to Cube Pruning to the proposed Algorithm 2 and Algorithm 3, and straightforwardly deduce the approximately monotonic version of the two algorithms. Accordingly, the proposed algorithms use a margin parameter  $\epsilon$  to control the uncertainty in finding the  $\text{best-}k$ . As with Cube Pruning, the proposed algorithms output an approximate  $\tilde{\text{best-}k}$  list in the approximately monotonic case. Since  $\Delta(\cdot)$  can be any function that satisfies the two stated properties, the proofs used to show that the three algorithms output

the same list in the monotonic search space case do not apply to the approximately monotonic case. Furthermore, Algorithm 2 could output different lists if used with different dimension orderings.

## 4. Experiments

In this section we test the algorithms presented, compare them with Cube Pruning, and empirically show that the lower complexity leads to a shorter execution time with no loss in accuracy. We implemented the proposed algorithms on top of a widely-used hierarchical Machine Translation system, cdec [3]. Implemented in C++, it is known to be one of the fastest decoders.

For the sake of comparability, the experiments were executed on the widely used NIST 2003 Chinese-English parallel corpus. The training corpus contains 239k sentence pairs with 6.9M Chinese words and 8.9M English words. A hierarchical phrase-based translation grammar was extracted using a suffix array rule extractor [7]. The NIST-03 test set is used for decoding from Chinese to English, which has 919 sentence pairs. The workstation used has Intel Core2 Duo CPU at 2.66 GHz with 4M of cache, 3.5 GB RAM, and is running Linux kernel version 2.6.24 and gcc version 4.2.4. The decoder uses SRI’s Language Model Toolkit version 1.5.11 [9]. All algorithms were configured to use a limit of 30 candidates at each node and no further pruning. Since in the cdec system uses  $\epsilon = 0$ , we use this setting for all the experiments.

Table 1: Total Time ( $T$ ) and average Time per-sentence, in seconds, reduction of execution Time relative to Cube Pruning, in percentage, and BLEU score.

Decoder	total T	sent T	T reduction	BLEU
CP	340.4	0.370	-	32.195
Alg. 2	322.7	0.351	5.2%	32.207
Alg. 3	325.1	0.354	4.5%	32.144

Table 1 reports results on decoding speed, comparing standard Cube Pruning with the two algorithms we propose. The measures reported are obtained by averaging a set of 20 experiments for each algorithm. We can see that the best performance in time and BLEU score is achieved by Algorithm 2, although the differences in BLEU among the 3 results are not large enough to be of any consequence. On the other hand, the execution times of the three algorithms do show real differences. Algorithm 3 achieves a reduction of 4.5% on the total running time. Algorithm 2 achieves a higher reduction of 5.2%.

As described in [3], cdec is designed as a pipeline; the Cube Pruning is just a single module of the sequence of operations that make up the Machine Translation decoding. To have a clearer idea of the magnitude of the speedup obtained, we measured the time for the single module that contains the actual scoring of the translation hypergraph and that is changed to modify Cube Pruning into Algorithms 2 and 3.

In Table 2 we report those decoding speed measures. As we

Table 2: Measures for the Cube Pruning Step of the cdec pipeline. Total time and sentence time in seconds. Time reduction and translation score variation for the proposed algorithms in comparison with Cube Pruning.

Decoder	total T	sent. T	T reduction	score var.
CP	194.1	0.211	-	-
Alg. 2	171.8	0.187	11.47%	-0.015%
Alg. 3	177.9	0.193	8.33%	-0.024%

can see, the time reductions are larger in this case. Algorithm 2 executes in 11.47% less time than the original Cube Pruning version. Furthermore we report the average variation of the translation sentences score. The irrelevance of the variation shows that Alg. 2 and Alg. 3 do not make more search errors than CP and therefore show again the equivalence of the quality of the output of the three algorithms.

These results confirm our hypothesis that Algorithms 2 and 3 are faster because they consider fewer candidates. Considering fewer candidates could make these algorithms more susceptible to nonmonotonicity, but the very small variations in scores suggest that this is not an issue. Algorithm 3 considers the fewest candidates, but Algorithms 2 is slightly faster, indicating that the reduction in time spent manipulating data structures has a greater effect, for this application.

## 5. Discussion

As stated in section 3, the proofs used to show that the three algorithms output the same list in the monotonic search space case do not apply at the approximately monotonic case. The empirical results reported in Table 1 show that the three algorithms reach different BLEU scores given the same input. Therefore for the approximately monotonic case we can state that, theoretically and empirically, the three algorithms do not output the same list. Despite this difference, the BLEU score results are very close.

To understand how the lists produced by the three algorithms differ at each node of the hypergraph at decoding time, consider the Cube Pruning algorithm. In the approximately monotonic search space, it can happen that some element  $\mathbf{v}$  is popped from  $Q$  and moved to  $\tilde{\text{best-}k}$  before all its ancestors have been moved to  $\tilde{\text{best-}k}$ :  $\mathcal{A}_{\mathbf{v}} \not\subseteq \tilde{\text{best-}k}$ . Also it can happen that some of the ancestors of  $\mathbf{v}$  have not yet been considered when  $\mathbf{v}$  is moved to  $\tilde{\text{best-}k}$ :  $\mathcal{A}_{\mathbf{v}} \not\subseteq Q \cup \tilde{\text{best-}k}$ . These 2 cases cannot happen in the monotonic case, and are due to the perturbations of the search space introduced by  $\Delta(\cdot)$ . When an element  $\mathbf{v}$  is better than some of its ancestors, Cube Pruning can move it to the  $\tilde{\text{best-}k}$  anyway. Algorithm 2 and Algorithm 3, treat these cases differently. Algorithm 2 cannot add  $\mathbf{v}$  to  $\tilde{\text{best-}k}$  before all his ancestors are already in it. Algorithm 2 sometimes behaves like Cube Pruning and sometimes behaves like Algorithm 3, depending on the order we have given to the coordinates. These cases are the cause

of the divergence of behaviour between the three algorithms. By measuring how often these cases occur during Cube Pruning decoding, we can have an upper bound on the number of elements that could lead to a divergence of the proposed algorithms in the approximately monotonic case. In Table 3 we report the percentage of the occurrence of these events during Cube Pruning decoding of the NIST-03 test set.

Table 3: *Percentage of events measured during Cube Pruning decoding that could lead to divergence of decoding for the three algorithms.*

Measured:	$\mathcal{A}_v \not\subseteq Q \cup \tilde{\text{best}}\text{-}k$	$\mathcal{A}_v \not\subseteq \tilde{\text{best}}\text{-}k$
While decoding	0.65%	2.07%
At the end	0.24%	0.74%

While decoding, 2.07% of the elements were added to  $\tilde{\text{best}}\text{-}k$  before all their ancestors were added to  $\tilde{\text{best}}\text{-}k$ . And only the 0.65% of the elements were added to  $\tilde{\text{best}}\text{-}k$  before all their ancestors have been considered as candidates in  $Q$ . Any of these events may lead to a divergence in decoding, but even in these cases the divergence may not occur. For example, Algorithm 2 can behave like Cube Pruning in certain cases. Furthermore even if the divergence happens, it may be that later the algorithms reach the same state anyway, just inserting the same elements to  $\tilde{\text{best}}\text{-}k$  in a different order. Thus these percentages can be considered as upper bounds of the events that lead to an actual difference in output. The percentages of the occurrences of the same events measured on the returned list can be considered as a lower bound. The second line of Table 3 shows these measures. In conclusion, we can state that the percentage of elements that will be different in the returned list of Algorithm 3 will be between 2.07% and 0.74%. For Algorithm 2 we have the same upper bound of 2.07%, but since in some cases it can behave as Cube Pruning, it can happen that some of the elements that don't have all their parent elements in the  $\tilde{\text{best}}\text{-}k$  list may appear in the final list of Algorithm 2.

For those few elements that differ in the returned lists of the three algorithms, they will tend to be at the bottom of their respective  $\tilde{\text{best}}\text{-}k$  lists. Consider the general case, where algorithm A return  $u$  and algorithm B instead returns the worse-scoring  $v$ . From the proofs for the monotonic case, we know that this can only arise because there is at least one ancestor  $w$  of  $u$  which has blocked algorithm B from considering  $u$  because  $w$  is worse than all  $k$  elements in the  $\tilde{\text{best}}\text{-}k$  list for algorithm B. Thus  $w$  is not in the true  $\text{best}\text{-}k$ , and  $w$  is worse than  $v$ , which is in turn worse than  $u$  by assumption. Because nonmonotonicity is generally small, the difference in score between  $w$  and its descendant  $u$  is generally small, and therefore the difference in score between  $w$  and  $v$  is also generally small. Since  $w$  is not in the true  $\text{best}\text{-}k$  list, we can therefore conclude that both  $u$  and  $v$  are generally below or near the bottom of the true  $\text{best}\text{-}k$  list. Therefore we expect that both  $u$  and  $v$  will tend to be near the bottom of their

respective  $\tilde{\text{best}}\text{-}k$  lists.

## 6. Conclusions

We presented 2 improved versions of the Cube Pruning algorithm that leverage a more aggressive pruning to achieve lower complexity and faster execution time at zero cost in terms of accuracy. Indeed we prove that they have exactly the same performance in a monotonic search space, and we show empirically that for approximately monotonic search spaces, as in the case of Machine Translation with Language Model features, the accuracy remains comparable. These improvements in speed at zero cost can be critical in real time applications, where even a 10% time reduction can make a real difference.

## 7. Acknowledgments

This work was partly funded by Swiss NSF grant CRSI22\_127510 (COMTIS) and European Community FP7 grant 216594 (CLASSiC, [www.classic-project.org](http://www.classic-project.org)).

## 8. References

- [1] Chiang, D., A hierarchical phrase-based model for statistical machine translation, In Proceedings of the ACL 2005, Ann Arbor, Michigan.
- [2] Chiang, D., Hierarchical phrase-based translation, Computational Linguistics, 33(2):201-228, 2007.
- [3] Dyer, C. and Lopez, A. and Ganitkevitch, J. and Weese, J. and Ture, F. and Blunsom, P. and Setiawan, H. and Eidelman, V. and Resnik P., cdec: A Decoder, Alignment, and Learning framework for finite-state and context-free translation models, In Proceedings of the ACL 2010, Uppsala, Sweden.
- [4] Goodman, J., Semiring parsing, Computational Linguistics, 25, 573-605.
- [5] Hopkins, M. and Langmead, G., Cube Pruning as Heuristic Search, In Proceedings of the EMNLP 2009, Singapore.
- [6] Huang, L. and Chiang, D., Better k-best Parsing, In Proceedings of the IWPT 2005, Vancouver, Canada.
- [7] Lopez, A., Hierarchical Phrase-Based Translation with Suffix Arrays, In Proceedings of the EMNLP 2007, Prague, Czech Republic.
- [8] Riesa, J. and Marcu, D., Hierarchical Search for Word Alignment, In Proceedings of the ACL 2010, Uppsala, Sweden.
- [9] Stolcke, A., SRILM - An extensible language modeling toolkit In Proceedings of the International Conference on Spoken Language Processing 2002, Denver, CO.