Exact Decoding of Syntactic Translation Models through Lagrangian Relaxation

Alexander M. Rush
MIT CSAIL,
Cambridge, MA 02139, USA
srush@csail.mit.edu

Michael Collins
Department of Computer Science,
Columbia University,
New York, NY 10027, USA
mcollins@cs.columbia.edu

Abstract

We describe an exact decoding algorithm for syntax-based statistical translation. The approach uses Lagrangian relaxation to decompose the decoding problem into tractable subproblems, thereby avoiding exhaustive dynamic programming. The method recovers exact solutions, with certificates of optimality, on over 97% of test examples; it has comparable speed to state-of-the-art decoders.

1 Introduction

Recent work has seen widespread use of synchronous probabilistic grammars in statistical machine translation (SMT). The decoding problem for a broad range of these systems (e.g., (Chiang, 2005; Marcu et al., 2006; Shen et al., 2008)) corresponds to the intersection of a (weighted) hypergraph with an n-gram language model.[1] The hypergraph represents a large set of possible translations, and is created by applying a synchronous grammar to the source language string. The language model is then used to rescore the translations in the hypergraph.

Decoding with these models is challenging, largely because of the cost of integrating an n-gram language model into the search process. Exact dynamic programming algorithms for the problem are well known (Bar-Hillel et al., 1964), but are too expensive to be used in practice.[2] Previous work on decoding for syntax-based SMT has therefore been focused primarily on approximate search methods.

This paper describes an efficient algorithm for exact decoding of synchronous grammar models for translation. We avoid the construction of (Bar-Hillel et al., 1964) by using *Lagrangian relaxation* to decompose the decoding problem into the following sub-problems:

1. Dynamic programming over the weighted hypergraph. This step does not require language model integration, and hence is highly efficient.

2. Application of an all-pairs shortest path algorithm to a directed graph derived from the weighted hypergraph. The size of the derived directed graph is linear in the size of the hypergraph, hence this step is again efficient.

Informally, the first decoding algorithm incorporates the weights and hard constraints on translations from the synchronous grammar, while the second decoding algorithm is used to integrate language model scores. Lagrange multipliers are used to enforce agreement between the structures produced by the two decoding algorithms.

In this paper we first give background on hypergraphs and the decoding problem. We then describe our decoding algorithm. The algorithm uses a subgradient method to minimize a dual function. The dual corresponds to a particular linear programming (LP) relaxation of the original decoding problem. The method will recover an exact solution, with a certificate of optimality, if the underlying LP relaxation has an integral solution. In some cases, however, the underlying LP will have a fractional solution, in which case the method will not be exact. The second technical contribution of this paper is to describe a method that iteratively tightens the underlying LP relaxation until an exact solution is produced. We do this by gradually introducing constraints to step 1 (dynamic programming over the hypergraph), while still maintaining efficiency.

[1] This problem is also relevant to other areas of statistical NLP, for example NL generation (Langkilde, 2000).

[2] E.g., with a trigram language model they run in $O(|E|w^6)$ time, where $|E|$ is the number of edges in the hypergraph, and w is the number of distinct lexical items in the hypergraph.

Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics, pages 72–82,
Portland, Oregon, June 19-24, 2011. ©2011 Association for Computational Linguistics

We report experiments using the tree-to-string model of (Huang and Mi, 2010). Our method gives exact solutions on over 97% of test examples. The method is comparable in speed to state-of-the-art decoding algorithms; for example, over 70% of the test examples are decoded in 2 seconds or less. We compare our method to cube pruning (Chiang, 2007), and find that our method gives improved model scores on a significant number of examples. One consequence of our work is that we give accurate estimates of the number of search errors for cube pruning.

2 Related Work

A variety of approximate decoding algorithms have been explored for syntax-based translation systems, including cube-pruning (Chiang, 2007; Huang and Chiang, 2007), left-to-right decoding with beam search (Watanabe et al., 2006; Huang and Mi, 2010), and coarse-to-fine methods (Petrov et al., 2008).

Recent work has developed decoding algorithms based on finite state transducers (FSTs). Iglesias et al. (2009) show that exact FST decoding is feasible for a phrase-based system with limited reordering (the MJ1 model (Kumar and Byrne, 2005)), and de Gispert et al. (2010) show that exact FST decoding is feasible for a specific class of hierarchical grammars (shallow-1 grammars). Approximate search methods are used for more complex reordering models or grammars. The FST algorithms are shown to produce higher scoring solutions than cube-pruning on a large proportion of examples.

Lagrangian relaxation is a classical technique in combinatorial optimization (Korte and Vygen, 2008). Lagrange multipliers are used to add linear constraints to an existing problem that can be solved using a combinatorial algorithm; the resulting dual function is then minimized, for example using subgradient methods. In recent work, *dual decomposition*—a special case of Lagrangian relaxation, where the linear constraints enforce agreement between two or more models—has been applied to inference in Markov random fields (Wainwright et al., 2005; Komodakis et al., 2007; Sontag et al., 2008), and also to inference problems in NLP (Rush et al., 2010; Koo et al., 2010). There are close connections between dual decomposition and work on belief propagation (Smith and Eisner, 2008).

3 Background: Hypergraphs

Translation with many syntax-based systems (e.g., (Chiang, 2005; Marcu et al., 2006; Shen et al., 2008; Huang and Mi, 2010)) can be implemented as a two-step process. The first step is to take an input sentence in the source language, and from this to create a hypergraph (sometimes called a translation forest) that represents the set of possible translations (strings in the target language) and derivations under the grammar. The second step is to integrate an n-gram language model with this hypergraph. For example, in the system of (Chiang, 2005), the hypergraph is created as follows: first, the source side of the synchronous grammar is used to create a parse forest over the source language string. Second, transduction operations derived from synchronous rules in the grammar are used to create the target-language hypergraph. Chiang's method uses a synchronous context-free grammar, but the hypergraph formalism is applicable to a broad range of other grammatical formalisms, for example dependency grammars (e.g., (Shen et al., 2008)).

A hypergraph is a pair (V, E) where $V = \{1, 2, \ldots, |V|\}$ is a set of vertices, and E is a set of hyperedges. A single distinguished vertex is taken as the root of the hypergraph; without loss of generality we take this vertex to be $v = 1$. Each hyperedge $e \in E$ is a tuple $\langle\langle v_1, v_2, \ldots, v_k\rangle, v_0\rangle$ where $v_0 \in V$, and $v_i \in \{2 \ldots |V|\}$ for $i = 1 \ldots k$. The vertex v_0 is referred to as the *head* of the edge. The ordered sequence $\langle v_1, v_2, \ldots, v_k\rangle$ is referred to as the *tail* of the edge; in addition, we sometimes refer to $v_1, v_2, \ldots v_k$ as the *children* in the edge. The number of children k may vary across different edges, but $k \geq 1$ for all edges (i.e., each edge has at least one child). We will use $h(e)$ to refer to the head of an edge e, and $t(e)$ to refer to the tail.

We will assume that the hypergraph is acyclic: intuitively this will mean that no derivation (as defined below) contains the same vertex more than once (see (Martin et al., 1990) for a formal definition).

Each vertex $v \in V$ is either a *non-terminal* in the hypergraph, or a *leaf*. The set of non-terminals is

$$V_N = \{v \in V : \exists e \in E \quad \text{such that} \quad h(e) = v\}$$

Conversely, the set of leaves is defined as

$$V_L = \{v \in V : \nexists e \in E \quad \text{such that} \quad h(e) = v\}$$

Finally, we assume that each $v \in V$ has a label $l(v)$. The labels for leaves will be *words*, and will be important in defining strings and language model scores for those strings. The labels for non-terminal nodes will not be important for results in this paper.[3]

We now turn to derivations. Define an *index set* $\mathcal{I} = V \cup E$. A derivation is represented by a vector $y = \{y_r : r \in \mathcal{I}\}$ where $y_v = 1$ if vertex v is used in the derivation, $y_v = 0$ otherwise (similarly $y_e = 1$ if edge e is used in the derivation, $y_e = 0$ otherwise). Thus y is a vector in $\{0,1\}^{|\mathcal{I}|}$. A valid derivation satisfies the following constraints:

- $y_1 = 1$ (the root must be in the derivation).
- For all $v \in V_N$, $y_v = \sum_{e:h(e)=v} y_e$.
- For all $v \in 2 \ldots |V|$, $y_v = \sum_{e:v \in t(e)} y_e$.

We use $\mathcal{Y}$ to refer to the set of valid derivations. The set $\mathcal{Y}$ is a subset of $\{0,1\}^{|\mathcal{I}|}$ (not all members of $\{0,1\}^{|\mathcal{I}|}$ will correspond to valid derivations).

Each derivation y in the hypergraph will imply an ordered sequence of leaves $v_1 \ldots v_n$. We use $s(y)$ to refer to this sequence. The *sentence* associated with the derivation is then $l(v_1) \ldots l(v_n)$.

In a weighted hypergraph problem, we assume a parameter vector $\theta = \{\theta_r : r \in \mathcal{I}\}$. The score for any derivation is $f(y) = \theta \cdot y = \sum_{r \in \mathcal{I}} \theta_r y_r$. Simple bottom-up dynamic programming—essentially the CKY algorithm—can be used to find $y^* = \arg\max_{y \in \mathcal{Y}} f(y)$ under these definitions.

The focus of this paper will be to solve problems involving the integration of a k'th order language model with a hypergraph. In these problems, the score for a derivation is modified to be

$$f(y) = \sum_{r \in \mathcal{I}} \theta_r y_r + \sum_{i=k}^{n} \theta(v_{i-k+1}, v_{i-k+2}, \ldots, v_i) \quad (1)$$

where $v_1 \ldots v_n = s(y)$. The $\theta(v_{i-k+1}, \ldots, v_i)$ parameters score n-grams of length k. These parameters are typically defined by a language model, for example with $k = 3$ we would have $\theta(v_{i-2}, v_{i-1}, v_i) = \log p(l(v_i)|l(v_{i-2}), l(v_{i-1}))$. The problem is then to find $y^* = \arg\max_{y \in \mathcal{Y}} f(y)$ under this definition.

Throughout this paper we make the following assumption when using a bigram language model:

[3] They might for example be non-terminal symbols from the grammar used to generate the hypergraph.

Assumption 3.1 *(*Bigram start/end assumption.*) For any derivation y, with leaves $s(y) = v_1, v_2, \ldots, v_n$, it is the case that: (1) $v_1 = 2$ and $v_n = 3$; (2) the leaves 2 and 3 cannot appear at any other position in the strings $s(y)$ for $y \in \mathcal{Y}$; (3) $l(2) =$ <s> where <s> is the start symbol in the language model; (4) $l(3) =$ </s> where </s> is the end symbol.*

This assumption allows us to incorporate language model terms that depend on the start and end symbols. It also allows a clean solution for boundary conditions (the start/end of strings).[4]

4 A Simple Lagrangian Relaxation Algorithm

We now give a Lagrangian relaxation algorithm for integration of a hypergraph with a bigram language model, in cases where the hypergraph satisfies the following simplifying assumption:

Assumption 4.1 *(*The strict ordering assumption.*) For any two leaves v and w, it is either the case that: 1) for all derivations y such that v and w are both in the sequence $l(y)$, v precedes w; or 2) for all derivations y such that v and w are both in $l(y)$, w precedes v.*

Thus under this assumption, the relative ordering of any two leaves is fixed. This assumption is overly restrictive:[5] the next section describes an algorithm that does not require this assumption. However deriving the simple algorithm will be useful in developing intuition, and will lead directly to the algorithm for the unrestricted case.

4.1 A Sketch of the Algorithm

At a high level, the algorithm is as follows. We introduce Lagrange multipliers $u(v)$ for all $v \in V_L$, with initial values set to zero. The algorithm then involves the following steps: (1) For each leaf v, find the previous leaf w that maximizes the score $\theta(w, v) - u(w)$ (call this leaf $\alpha^*(v)$, and define $\alpha_v = \theta(\alpha^*(v), v) - u(\alpha^*(v))$). (2) find the highest scoring derivation using dynamic programming

[4] The assumption generalizes in the obvious way to k'th order language models: e.g., for trigram models we assume that $v_1 = 2$, $v_2 = 3$, $v_n = 4$, $l(2) = l(3) =$ <s>, $l(4) =$ </s>.

[5] It is easy to come up with examples that violate this assumption: for example a hypergraph with edges $\langle\langle 4, 5\rangle, 1\rangle$ and $\langle\langle 5, 4\rangle, 1\rangle$ violates the assumption. The hypergraphs found in translation frequently contain alternative orderings such as this.

over the original (non-intersected) hypergraph, with leaf nodes having weights $\theta_v + \alpha_v + u(v)$. (3) If the output derivation from step 2 has the same set of bigrams as those from step 1, then we have an exact solution to the problem. Otherwise, the Lagrange multipliers $u(v)$ are modified in a way that encourages agreement of the two steps, and we return to step 1.

Steps 1 and 2 can be performed efficiently; in particular, we avoid the classical dynamic programming intersection, instead relying on dynamic programming over the original, simple hypergraph.

4.2 A Formal Description

We now give a formal description of the algorithm. Define $\mathcal{B} \subseteq V_L \times V_L$ to be the set of all ordered pairs $\langle v, w\rangle$ such that there is at least one derivation y with v directly preceding w in $s(y)$. Extend the bit-vector y to include variables $y(v,w)$ for $\langle v, w\rangle \in \mathcal{B}$ where $y(v,w) = 1$ if leaf v is followed by w in $s(y)$, 0 otherwise. We redefine the index set to be $\mathcal{I} = V \cup E \cup \mathcal{B}$, and define $\mathcal{Y} \subseteq \{0,1\}^{|\mathcal{I}|}$ to be the set of all possible derivations. Under assumptions 3.1 and 4.1 above, $\mathcal{Y} = \{y : y$ satisfies constraints **C0**, **C1**, **C2**$\}$ where the constraint definitions are:

- **(C0)** The y_v and y_e variables form a derivation in the hypergraph, as defined in section 3.
- **(C1)** For all $v \in V_L$ such that $v \neq 2$, $y_v = \sum_{w:\langle w,v\rangle \in \mathcal{B}} y(w,v)$.
- **(C2)** For all $v \in V_L$ such that $v \neq 3$, $y_v = \sum_{w:\langle v,w\rangle \in \mathcal{B}} y(v,w)$.

C1 states that each leaf in a derivation has exactly one in-coming bigram, and that each leaf not in the derivation has 0 incoming bigrams; **C2** states that each leaf in a derivation has exactly one out-going bigram, and that each leaf not in the derivation has 0 outgoing bigrams.[6]

The score of a derivation is now $f(y) = \theta \cdot y$, i.e.,

$$f(y) = \sum_v \theta_v y_v + \sum_e \theta_e y_e + \sum_{\langle v,w\rangle \in \mathcal{B}} \theta(v,w) y(v,w)$$

where $\theta(v,w)$ are scores from the language model. Our goal is to compute $y^* = \arg\max_{y \in \mathcal{Y}} f(y)$.

[6]Recall that according to the bigram start/end assumption the leaves 2/3 are reserved for the start/end of the sequence $s(y)$, and hence do not have an incoming/outgoing bigram.

Initialization: Set $u^0(v) = 0$ for all $v \in V_L$
Algorithm: For $t = 1 \dots T$:

- $y^t = \arg\max_{y \in \mathcal{Y}'} L(u^{t-1}, y)$
- **If** y^t satisfies constraints **C2**, **return** y^t,
  Else $\forall v \in V_L$, $u^t(v) =$
  $u^{t-1}(v) - \delta^t \left(y^t(v) - \sum_{w:\langle v,w\rangle \in \mathcal{B}} y^t(v,w) \right)$.

Figure 1: A simple Lagrangian relaxation algorithm. $\delta^t > 0$ is the step size at iteration t.

Next, define $\mathcal{Y}'$ as

$$\mathcal{Y}' = \{y : y \text{ satisfies constraints } \textbf{C0} \text{ and } \textbf{C1}\}$$

In this definition we have dropped the **C2** constraints. To incorporate these constraints, we use Lagrangian relaxation, with one Lagrange multiplier $u(v)$ for each constraint in **C2**. The Lagrangian is

$$\begin{aligned} L(u,y) &= f(y) + \sum_v u(v)(y(v) - \sum_{w:\langle v,w\rangle \in \mathcal{B}} y(v,w)) \\ &= \beta \cdot y \end{aligned}$$

where $\beta_v = \theta_v + u(v)$, $\beta_e = \theta_e$, and $\beta(v,w) = \theta(v,w) - u(v)$.

The dual problem is to find $\min_u L(u)$ where

$$L(u) = \max_{y \in \mathcal{Y}'} L(u,y)$$

Figure 1 shows a *subgradient* method for solving this problem. At each point the algorithm finds $y^t = \arg\max_{y \in \mathcal{Y}'} L(u^{t-1}, y)$, where u^{t-1} are the Lagrange multipliers from the previous iteration. If y^t satisfies the **C2** constraints in addition to **C0** and **C1**, then it is returned as the output from the algorithm. Otherwise, the multipliers $u(v)$ are updated. Intuitively, these updates encourage the values of y_v and $\sum_{w:\langle v,w\rangle \in \mathcal{B}} y(v,w)$ to be equal; formally, these updates correspond to subgradient steps.

The main computational step at each iteration is to compute $\arg\max_{y \in \mathcal{Y}'} L(u^{t-1}, y)$ This step is easily solved, as follows (we again use β_v, β_e and $\beta(v_1, v_2)$ to refer to the parameter values that incorporate Lagrange multipliers):

- For all $v \in V_L$, define $\alpha^*(v) = \arg\max_{w:\langle w,v\rangle \in \mathcal{B}} \beta(w,v)$ and $\alpha_v = \beta(\alpha^*(v), v)$. For all $v \in V_N$ define $\alpha_v = 0$.

- Using dynamic programming, find values for the y_v and y_e variables that form a valid derivation, and that maximize
$f'(y) = \sum_v(\beta_v + \alpha_v)y_v + \sum_e \beta_e y_e$.

- Set $y(v, w) = 1$ iff $y(w) = 1$ and $\alpha^*(w) = v$.

The critical point here is that through our definition of $\mathcal{Y}'$, which ignores the **C2** constraints, we are able to do efficient search as just described. In the first step we compute the highest scoring incoming bigram for each leaf v. In the second step we use conventional dynamic programming over the hypergraph to find an optimal derivation that incorporates weights from the first step. Finally, we fill in the $y(v, w)$ values. Each iteration of the algorithm runs in $O(|E| + |\mathcal{B}|)$ time.

There are close connections between Lagrangian relaxation and linear programming relaxations. The most important formal results are: 1) for any value of u, $L(u) \geq f(y^*)$ (hence the dual value provides an upper bound on the optimal primal value); 2) under an appropriate choice of the step sizes δ^t, the subgradient algorithm is guaranteed to converge to the minimum of $L(u)$ (i.e., we will minimize the upper bound, making it as tight as possible); 3) if at any point the algorithm in figure 1 finds a y^t that satisfies the **C2** constraints, then this is guaranteed to be the optimal primal solution.

Unfortunately, this algorithm may fail to produce a good solution for hypergraphs where the strict ordering constraint does not hold. In this case it is possible to find derivations y that satisfy constraints **C0**, **C1**, **C2**, but which are invalid. As one example, consider a derivation with $s(y) = 2, 4, 5, 3$ and $y(2, 3) = y(4, 5) = y(5, 4) = 1$. The constraints are all satisfied in this case, but the bigram variables are invalid (e.g., they contain a cycle).

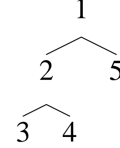
5 The Full Algorithm

We now describe our full algorithm, which does not require the strict ordering constraint. In addition, the full algorithm allows a trigram language model. We first give a sketch, and then give a formal definition.

5.1 A Sketch of the Algorithm

A crucial idea in the new algorithm is that of *paths* between leaves in hypergraph derivations. Previously, for each derivation y, we had defined $s(y) = v_1, v_2, \ldots, v_n$ to be the sequence of leaves in y. In addition, we will define $g(y) = p_0, v_1, p_1, v_2, p_2, v_3, p_3, \ldots, p_{n-1}, v_n, p_n$ where each p_i is a path in the derivation between leaves v_i and v_{i+1}. The path traces through the non-terminals that are between the two leaves in the tree.

As an example, consider the following derivation (with hyperedges $\langle\langle 2, 5\rangle, 1\rangle$ and $\langle\langle 3, 4\rangle, 2\rangle$):



For this example $g(y)$ is $\langle 1\downarrow, 2\downarrow\rangle$ $\langle 2\downarrow, 3\downarrow\rangle$ $\langle 3\downarrow\rangle$, 3, $\langle 3\uparrow\rangle$ $\langle 3\uparrow, 4\downarrow\rangle$ $\langle 4\downarrow\rangle$, 4, $\langle 4\uparrow\rangle$ $\langle 4\uparrow, 2\uparrow\rangle$ $\langle 2\uparrow, 5\downarrow\rangle$ $\langle 5\downarrow\rangle$, 5, $\langle 5\uparrow\rangle$ $\langle 5\uparrow, 1\uparrow\rangle$. States of the form $\langle a\downarrow\rangle$ and $\langle a\uparrow\rangle$ where a is a leaf appear in the paths respectively before/after the leaf a. States of the form $\langle a, b\rangle$ correspond to the steps taken in a top-down, left-to-right, traversal of the tree, where down and up arrows indicate whether a node is being visited for the first or second time (the traversal in this case would be $1, 2, 3, 4, 2, 5, 1$).

The mapping from a derivation y to a path $g(y)$ can be performed using the algorithm in figure 2. For a given derivation y, define $E(y) = \{y : y_e = 1\}$, and use $E(y)$ as the set of input edges to this algorithm. The output from the algorithm will be a set of states S, and a set of directed edges T, which together fully define the path $g(y)$.

In the simple algorithm, the first step was to predict the previous leaf for each leaf v, under a score that combined a language model score with a Lagrange multiplier score (i.e., compute $\arg\max_w \beta(w, v)$ where $\beta(w, v) = \theta(w, v) + u(w)$). In this section we describe an algorithm that for each leaf v again predicts the previous leaf, but in addition predicts the full *path* back to that leaf. For example, rather than making a prediction for leaf 5 that it should be preceded by leaf 4, we would also predict the path $\langle 4\uparrow\rangle\langle 4\uparrow, 2\uparrow\rangle$ $\langle 2\uparrow, 5\downarrow\rangle\langle 5\downarrow\rangle$ between these two leaves. Lagrange multipliers will be used to enforce consistency between these predictions (both paths and previous words) and a valid derivation.

Input: A set E of hyperedges. **Output:** A directed graph S, T where S is a set of vertices, and T is a set of edges.
Step 1: Creating S: Define $S = \cup_{e \in E} S(e)$ where $S(e)$ is defined as follows. Assume $e = \langle\langle v_1, v_2, \ldots, v_k\rangle, v_0\rangle$. Include the following states in $S(e)$: (1) $\langle v_0 \downarrow, v_1 \downarrow\rangle$ and $\langle v_k \uparrow, v_0 \uparrow\rangle$. (2) $\langle v_j \uparrow, v_{j+1} \downarrow\rangle$ for $j = 1 \ldots k-1$ (if $k = 1$ then there are no such states). (3) In addition, for any v_j for $j = 1 \ldots k$ such that $v_j \in V_L$, add the states $\langle v_j \downarrow\rangle$ and $\langle v_j \uparrow\rangle$.
Step 2: Creating T**:** T is formed by including the following directed arcs: (1) Add an arc from $\langle a, b\rangle \in S$ to $\langle c, d\rangle \in S$ whenever $b = c$. (2) Add an arc from $\langle a, b \downarrow\rangle \in S$ to $\langle c \downarrow\rangle \in S$ whenever $b = c$. (3) Add an arc from $\langle a \uparrow\rangle \in S$ to $\langle b \uparrow, c\rangle \in S$ whenever $a = b$.

Figure 2: Algorithm for constructing a directed graph (S, T) from a set of hyperedges E.

5.2 A Formal Description

We first use the algorithm in figure 2 with the entire set of hyperedges, E, as its input. The result is a directed graph (S, T) that contains *all possible paths* for valid derivations in V, E (it also contains additional, ill-formed paths). We then introduce the following definition:

Definition 5.1 A trigram path p *is* $p = \langle v_1, p_1, v_2, p_2, v_3\rangle$ *where: a)* $v_1, v_2, v_3 \in V_L$*; b)* p_1 *is a path (sequence of states) between nodes* $\langle v_1 \uparrow\rangle$ *and* $\langle v_2 \downarrow\rangle$ *in the graph* (S, T)*; c)* p_2 *is a path between nodes* $\langle v_2 \uparrow\rangle$ *and* $\langle v_3 \downarrow\rangle$ *in the graph* (S, T)*. We define* $\mathcal{P}$ *to be the set of all trigram paths in* (S, T).

The set $\mathcal{P}$ of trigram paths plays an analogous role to the set $\mathcal{B}$ of bigrams in our previous algorithm.

We use $v_1(p), p_1(p), v_2(p), p_2(p), v_3(p)$ to refer to the individual components of a path p. In addition, define S_N to be the set of states in S of the form $\langle a, b\rangle$ (as opposed to the form $\langle c \downarrow\rangle$ or $\langle c \uparrow\rangle$ where $c \in V_L$).

We now define a new index set, $\mathcal{I} = V \cup E \cup S_N \cup \mathcal{P}$, adding variables y_s for $s \in S_N$, and y_p for $p \in \mathcal{P}$. If we take $\mathcal{Y} \subset \{0, 1\}^{|\mathcal{I}|}$ to be the set of valid derivations, the optimization problem is to find $y^* = \arg\max_{y \in \mathcal{Y}} f(y)$, where $f(y) = \theta \cdot y$, that is,

$$f(y) = \sum_v \theta_v y_v + \sum_e \theta_e y_e + \sum_s \theta_s y_s + \sum_p \theta_p y_p$$

In particular, we might define $\theta_s = 0$ for all s, and $\theta_p = \log p(l(v_3(p))|l(v_1(p)), l(v_2(p)))$ where $p(w_3|w_1, w_2)$ is a trigram probability.

The set $\mathcal{P}$ is large (typically exponential in size): however, we will see that we do not need to represent the y_p variables explicitly. Instead we will be able to leverage the underlying structure of a path as a sequence of states.

The set of valid derivations is $\mathcal{Y} = \{y : y$ satisfies constraints **D0–D6**$\}$ where the constraints are shown in figure 3. **D1** simply states that $y_s = 1$ iff there is exactly one edge e in the derivation such that $s \in S(e)$. Constraints **D2–D4** enforce consistency between leaves in the trigram paths, and the y_v values. Constraints **D5** and **D6** enforce consistency between states seen in the paths, and the y_s values.

The Lagrangian relaxation algorithm is then derived in a similar way to before. Define

$$\mathcal{Y}' = \{y : y \text{ satisfies constraints } \textbf{D0–D2}\}$$

We have dropped the **D3–D6** constraints, but these will be introduced using Lagrange multipliers. The resulting Lagrangian is shown in figure 3, and can be written as $L(y, \lambda, \gamma, u, v) = \beta \cdot y$ where $\beta_v = \theta_v + \lambda_v + \gamma_v$, $\beta_s = \theta_s + u_s + v_s$, $\beta_p = \theta_p - \lambda(v_2(p)) - \gamma(v_1(p)) - \sum_{s \in p_1(p)} u(s) - \sum_{s \in p_2(p)} v(s)$.

The dual is $L(\lambda, \gamma, u, v) = \max_{y \in \mathcal{Y}'} L(y, \lambda, \gamma, u, v)$; figure 4 shows a subgradient method that minimizes this dual. The key step in the algorithm at each iteration is to compute

• **D0.** The y_v and y_e variables form a valid derivation in the original hypergraph.
• **D1.** For all $s \in S_N$, $y_s = \sum_{e : s \in S(e)} y_e$ (see figure 2 for the definition of $S(e)$).
• **D2.** For all $v \in V_L$, $y_v = \sum_{p : v_3(p) = v} y_p$
• **D3.** For all $v \in V_L$, $y_v = \sum_{p : v_2(p) = v} y_p$
• **D4.** For all $v \in V_L$, $y_v = \sum_{p : v_1(p) = v} y_p$
• **D5.** For all $s \in S_N$, $y_s = \sum_{p : s \in p_1(p)} y_p$
• **D6.** For all $s \in S_N$, $y_s = \sum_{p : s \in p_2(p)} y_p$
• Lagrangian with Lagrange multipliers for **D3–D6**:

$$\begin{aligned} L(y, \lambda, \gamma, u, v) = \theta \cdot y & \\ + \sum_v \lambda_v \Big(y_v - \sum_{p : v_2(p) = v} y_p\Big) & \\ + \sum_v \gamma_v \Big(y_v - \sum_{p : v_1(p) = v} y_p\Big) & \\ + \sum_s u_s \Big(y_s - \sum_{p : s \in p_1(p)} y_p\Big) & \\ + \sum_s v_s \Big(y_s - \sum_{p : s \in p_2(p)} y_p\Big). & \end{aligned}$$

Figure 3: Constraints **D0–D6**, and the Lagrangian.

Initialization: Set $\lambda^0 = 0, \gamma^0 = 0, u^0 = 0, v^0 = 0$
Algorithm: For $t = 1 \dots T$:

- $y^t = \arg\max_{y \in \mathcal{Y}'} L(y, \lambda^{t-1}, \gamma^{t-1}, u^{t-1}, v^{t-1})$
- If y^t satisfies the constraints **D3–D6**, return y^t, else:
  - $\forall v \in V_L, \lambda_v^t = \lambda_v^{t-1} - \delta^t (y_v^t - \sum_{p: v_2(p) = v} y_p^t)$
  - $\forall v \in V_L, \gamma_v^t = \gamma_v^{t-1} - \delta^t (y_v^t - \sum_{p: v_1(p) = v} y_p^t)$
  - $\forall s \in S_N, u_s^t = u_s^{t-1} - \delta^t (y_s^t - \sum_{p: s \in p_1(p)} y_p^t)$
  - $\forall s \in S_N, v_s^t = v_s^{t-1} - \delta^t (y_s^t - \sum_{p: s \in p_2(p)} y_p^t)$

Figure 4: The full Lagrangian relaxation algortihm. $\delta^t > 0$ is the step size at iteration t.

$\arg\max_{y \in \mathcal{Y}'} L(y, \lambda, \gamma, u, v) = \arg\max_{y \in \mathcal{Y}'} \beta \cdot y$ where β is defined above. Again, our definition of $\mathcal{Y}'$ allows this maximization to be performed efficiently, as follows:

1. For each $v \in V_L$, define $\alpha_v^* = \arg\max_{p: v_3(p) = v} \beta(p)$, and $\alpha_v = \beta(\alpha_v^*)$. (i.e., for each v, compute the highest scoring trigram path ending in v.)

2. Find values for the y_v, y_e and y_s variables that form a valid derivation, and that maximize
$f'(y) = \sum_v (\beta_v + \alpha_v) y_v + \sum_e \beta_e y_e + \sum_s \beta_s y_s$

3. Set $y_p = 1$ iff $y_{v_3(p)} = 1$ and $p = \alpha_{v_3(p)}^*$.

The first step involves finding the highest scoring incoming trigram path for each leaf v. This step can be performed efficiently using the Floyd-Warshall all-pairs shortest path algorithm (Floyd, 1962) over the graph (S, T); the details are given in the appendix. The second step involves simple dynamic programming over the hypergraph (V, E) (it is simple to integrate the β_s terms into this algorithm). In the third step, the path variables y_p are filled in.

5.3 Properties

We now describe some important properties of the algorithm:

Efficiency. The main steps of the algorithm are: 1) construction of the graph (S, T); 2) at each iteration, dynamic programming over the hypergraph (V, E); 3) at each iteration, all-pairs shortest path algorithms over the graph (S, T). Each of these steps is vastly more efficient than computing an exact intersection of the hypergraph with a language model.

Exact solutions. By usual guarantees for Lagrangian relaxation, if at any point the algorithm returns a solution y^t that satisfies constraints **D3–D6**, then y^t exactly solves the problem in Eq. 1.

Upper bounds. At each point in the algorithm, $L(\lambda^t, \gamma^t, u^t, v^t)$ is an upper bound on the score of the optimal primal solution, $f(y^*)$. Upper bounds can be useful in evaluating the quality of primal solutions from either our algorithm or other methods such as cube pruning.

Simplicity of implementation. Construction of the (S, T) graph is straightforward. The other steps—hypergraph dynamic programming, and all-pairs shortest path—are widely known algorithms that are simple to implement.

6 Tightening the Relaxation

The algorithm that we have described minimizes the dual function $L(\lambda, \gamma, u, v)$. By usual results for Lagrangian relaxation (e.g., see (Korte and Vygen, 2008)), L is the dual function for a particular LP relaxation arising from the definition of $\mathcal{Y}'$ and the additional constaints **D3–D6**. In some cases the LP relaxation has an integral solution, in which case the algorithm will return an optimal solution y^t.[7] In other cases, when the LP relaxation has a fractional solution, the subgradient algorithm will still converge to the minimum of L, but the primal solutions y^t will move between a number of solutions.

We now describe a method that incrementally adds hard constraints to the set $\mathcal{Y}'$, until the method returns an exact solution. For a given $y \in \mathcal{Y}'$, for any v with $y_v = 1$, we can recover the previous two leaves (the trigram ending in v) from either the path variables y_p, or the hypergraph variables y_e. Specifically, define $v_{-1}(v, y)$ to be the leaf preceding v in the trigram path p with $y_p = 1$ and $v_3(p) = v$, and $v_{-2}(v, y)$ to be the leaf two positions before v in the trigram path p with $y_p = 1$ and $v_3(p) = v$. Similarly, define $v'_{-1}(v, y)$ and $v'_{-2}(v, y)$ to be the preceding two leaves under the y_e variables. If the method has not converged, these two trigram definitions may not be consistent. For a con-

[7] Provided that the algorithm is run for enough iterations for convergence.

sistent solution, we require $v_{-1}(v, y) = v'_{-1}(v, y)$ and $v_{-2}(v, y) = v'_{-2}(v, y)$ for all v with $y_v = 1$. Unfortunately, explicitly enforcing all of these constraints would require exhaustive dynamic programming over the hypergraph using the (Bar-Hillel et al., 1964) method, something we wish to avoid.

Instead, we enforce a weaker set of constraints, which require far less computation. Assume some function $\pi : V_L \rightarrow \{1, 2, \ldots q\}$ that partitions the set of leaves into q different partitions. Then we will add the following constraints to $\mathcal{Y}'$:

$$\pi(v_{-1}(v, y)) = \pi(v'_{-1}(v, y))$$
$$\pi(v_{-2}(v, y)) = \pi(v'_{-2}(v, y))$$

for all v such that $y_v = 1$. Finding $\arg\max_{y \in \mathcal{Y}'} \theta \cdot y$ under this new definition of $\mathcal{Y}'$ can be performed using the construction of (Bar-Hillel et al., 1964), with q different lexical items (for brevity we omit the details). This is efficient if q is small.[8]

The remaining question concerns how to choose a partition π that is effective in tightening the relaxation. To do this we implement the following steps: 1) run the subgradient algorithm until L is close to convergence; 2) then run the subgradient algorithm for m further iterations, keeping track of all pairs of leaf nodes that violate the constraints (i.e., pairs $a = v_{-1}(v, y)/b = v'_{-1}(v, y)$ or $a = v_{-2}(v, y)/b = v'_{-2}(v, y)$ such that $a \neq b$); 3) use a graph coloring algorithm to find a small partition that places all pairs $\langle a, b\rangle$ into separate partitions; 4) continue running Lagrangian relaxation, with the new constraints added. We expand π at each iteration to take into account new pairs $\langle a, b\rangle$ that violate the constraints.

In related work, Sontag et al. (2008) describe a method for inference in Markov random fields where additional constraints are chosen to tighten an underlying relaxation. Other relevant work in NLP includes (Tromble and Eisner, 2006; Riedel and Clarke, 2006). Our use of partitions π is related to previous work on coarse-to-fine inference for machine translation (Petrov et al., 2008).

7 Experiments

We report experiments on translation from Chinese to English, using the tree-to-string model described in (Huang and Mi, 2010). We use an identical model, and identical development and test data, to that used by Huang and Mi.[9] The translation model is trained on 1.5M sentence pairs of Chinese-English data; a trigram language model is used. The development data is the newswire portion of the 2006 NIST MT evaluation test set (616 sentences). The test set is the newswire portion of the 2008 NIST MT evaluation test set (691 sentences).

Time	%age (LR)	%age (DP)	%age (ILP)	%age (LP)
0.5s	37.5	10.2	8.8	21.0
1.0s	57.0	11.6	13.9	31.1
2.0s	72.2	15.1	21.1	45.9
4.0s	82.5	20.7	30.7	63.7
8.0s	88.9	25.2	41.8	78.3
16.0s	94.4	33.3	54.6	88.9
32.0s	97.8	42.8	68.5	95.2
Median time	0.79s	77.5s	12.1s	2.4s

Figure 5: Results showing percentage of examples that are decoded in less than t seconds, for $t = 0.5, 1.0, 2.0, \ldots, 32.0$. LR = Lagrangian relaxation; DP = exhaustive dynamic programming; ILP = integer linear programming; LP = linear programming (LP does not recover an exact solution). The (I)LP experiments were carried out using Gurobi, a high-performance commercial-grade solver.

We ran the full algorithm with the tightening method described in section 6. We ran the method for a limit of 200 iterations, hence some examples may not terminate with an exact solution. Our method gives exact solutions on 598/616 development set sentences (97.1%), and 675/691 test set sentences (97.7%).

In cases where the method does not converge within 200 iterations, we can return the best primal solution y^t found by the algorithm during those iterations. We can also get an upper bound on the difference $f(y^*) - f(y^t)$ using $\min_t L(u_t)$ as an upper bound on $f(y^*)$. Of the examples that did not converge, the worst example had a bound that was 1.4% of $f(y^t)$ (more specifically, $f(y^t)$ was -24.74, and the upper bound on $f(y^*) - f(y^t)$ was 0.34).

Figure 5 gives information on decoding time for our method and two other exact decoding methods: integer linear programming (using constraints **D0–D6**), and exhaustive dynamic programming using the construction of (Bar-Hillel et al., 1964). Our

[8] In fact in our experiments we use the original hypergraph to compute admissible outside scores for an exact A* search algorithm for this problem. We have found the resulting search algorithm to be very efficient.

[9] We thank Liang Huang and Haitao Mi for providing us with their model and data.

method is clearly the most efficient, and is comparable in speed to state-of-the-art decoding algorithms.

We also compare our method to cube pruning (Chiang, 2007; Huang and Chiang, 2007). We reimplemented cube pruning in C++, to give a fair comparison to our method. Cube pruning has a parameter, b, dictating the maximum number of items stored at each chart entry. With $b = 50$, our decoder finds higher scoring solutions on 50.5% of all examples (349 examples), the cube-pruning method gets a strictly higher score on only 1 example (this was one of the examples that did not converge within 200 iterations). With $b = 500$, our decoder finds better solutions on 18.5% of the examples (128 cases), cube-pruning finds a better solution on 3 examples. The median decoding time for our method is 0.79 seconds; the median times for cube pruning with $b = 50$ and $b = 500$ are 0.06 and 1.2 seconds respectively.

Our results give a very good estimate of the percentage of search errors for cube pruning. A natural question is how large b must be before exact solutions are returned on almost all examples. Even at $b = 1000$, we find that our method gives a better solution on 95 test examples (13.7%).

Figure 5 also gives a speed comparison of our method to a linear programming (LP) solver that solves the LP relaxation defined by constraints **D0**–**D6**. We still see speed-ups, in spite of the fact that our method is solving a harder problem (it provides integral solutions). The Lagrangian relaxation method, when run without the tightening method of section 6, is solving a dual of the problem being solved by the LP solver. Hence we can measure how often the tightening procedure is absolutely necessary, by seeing how often the LP solver provides a fractional solution. We find that this is the case on 54.0% of the test examples: the tightening procedure is clearly important. Inspection of the tightening procedure shows that the number of partitions required (the parameter q) is generally quite small: 59% of examples that require tightening require $q \leq 6$; 97.2% require $q \leq 10$.

8 Conclusion

We have described a Lagrangian relaxation algorithm for exact decoding of syntactic translation models, and shown that it is significantly more efficient than other exact algorithms for decoding tree-to-string models. There are a number of possible ways to extend this work. Our experiments have focused on tree-to-string models, but the method should also apply to Hiero-style syntactic translation models (Chiang, 2007). Additionally, our experiments used a trigram language model, however the constraints in figure 3 generalize to higher-order language models. Finally, our algorithm recovers the 1-best translation for a given input sentence; it should be possible to extend the method to find k-best solutions.

A Computing the Optimal Trigram Paths

For each $v \in V_L$, define $\alpha_v = \max_{p: v_3(p)=v} \beta(p)$, where $\beta(p) = h(v_1(p), v_2(p), v_3(p)) - \lambda_1(v_1(p)) - \lambda_2(v_2(p)) - \sum_{s \in p_1(p)} u(s) - \sum_{s \in p_2(p)} v(s)$. Here h is a function that computes language model scores, and the other terms involve Lagrange mulipliers. Our task is to compute α^*_v for all $v \in V_L$.

It is straightforward to show that the S, T graph is *acyclic*. This will allow us to apply shortest path algorithms to the graph, even though the weights $u(s)$ and $v(s)$ can be positive or negative.

For any pair $v_1, v_2 \in V_L$, define $\mathcal{P}(v_1, v_2)$ to be the set of paths between $\langle v_1 \uparrow \rangle$ and $\langle v_2 \downarrow \rangle$ in the graph S, T. Each path p gets a score $score_u(p) = -\sum_{s \in p} u(s)$. Next, define $p^*_u(v_1, v_2) = \arg\max_{p \in \mathcal{P}(v_1, v_2)} score_u(p)$, and $score^*_u(v_1, v_2) = score_u(p^*)$. We assume similar definitions for $p^*_v(v_1, v_2)$ and $score^*_v(v_1, v_2)$. The p^*_u and $score^*_u$ values can be calculated using an all-pairs shortest path algorithm, with weights $u(s)$ on nodes in the graph. Similarly, p^*_v and $score^*_v$ can be computed using all-pairs shortest path with weights $v(s)$ on the nodes.

Having calculated these values, define $\mathcal{T}(v)$ for any leaf v to be the set of trigrams (x, y, v) such that: 1) $x, y \in V_L$; 2) there is a path from $\langle x \uparrow \rangle$ to $\langle y \downarrow \rangle$ and from $\langle y \uparrow \rangle$ to $\langle v \downarrow \rangle$ in the graph S, T. Then we can calculate

$$\alpha_v = \max_{(x,y,v) \in \mathcal{T}(v)} (h(x, y, v) - \lambda_1(x) - \lambda_2(y) + p^*_u(x, y) + p^*_v(y, v))$$

in $O(|\mathcal{T}(v)|)$ time, by brute force search through the set $\mathcal{T}(v)$.

Acknowledgments Alexander Rush and Michael Collins were supported under the GALE program of the Defense Advanced Research Projects Agency, Contract No. HR0011-06-C-0022. Michael Collins was also supported by NSF grant IIS-0915176. We also thank the anonymous reviewers for very helpful comments; we hope to fully address these in an extended version of the paper.

References

Y. Bar-Hillel, M. Perles, and E. Shamir. 1964. On formal properties of simple phrase structure grammars. In *Language and Information: Selected Essays on their Theory and Application*, pages 116–150.

D. Chiang. 2005. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 263–270. Association for Computational Linguistics.

D. Chiang. 2007. Hierarchical phrase-based translation. *computational linguistics*, 33(2):201–228.

Adria de Gispert, Gonzalo Iglesias, Graeme Blackwood, Eduardo R. Banga, and William Byrne. 2010. Hierarchical Phrase-Based Translation with Weighted Finite-State Transducers and Shallow-n Grammars. In *Computational linguistics*, volume 36, pages 505–533.

Robert W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM*, 5:345.

Liang Huang and David Chiang. 2007. Forest rescoring: Faster decoding with integrated language models. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 144–151, Prague, Czech Republic, June. Association for Computational Linguistics.

Liang Huang and Haitao Mi. 2010. Efficient incremental decoding for tree-to-string translation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 273–283, Cambridge, MA, October. Association for Computational Linguistics.

Gonzalo Iglesias, Adrià de Gispert, Eduardo R. Banga, and William Byrne. 2009. Rule filtering by pattern for efficient hierarchical translation. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, pages 380–388, Athens, Greece, March. Association for Computational Linguistics.

N. Komodakis, N. Paragios, and G. Tziritas. 2007. MRF optimization via dual decomposition: Message-passing revisited. In *International Conference on Computer Vision*.

Terry Koo, Alexander M. Rush, Michael Collins, Tommi Jaakkola, and David Sontag. 2010. Dual decomposition for parsing with non-projective head automata. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1288–1298, Cambridge, MA, October. Association for Computational Linguistics.

B.H. Korte and J. Vygen. 2008. *Combinatorial optimization: theory and algorithms*. Springer Verlag.

Shankar Kumar and William Byrne. 2005. Local phrase reordering models for statistical machine translation. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 161–168, Vancouver, British Columbia, Canada, October. Association for Computational Linguistics.

I. Langkilde. 2000. Forest-based statistical sentence generation. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, pages 170–177. Morgan Kaufmann Publishers Inc.

Daniel Marcu, Wei Wang, Abdessamad Echihabi, and Kevin Knight. 2006. Spmt: Statistical machine translation with syntactified target language phrases. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 44–52, Sydney, Australia, July. Association for Computational Linguistics.

R.K. Martin, R.L. Rardin, and B.A. Campbell. 1990. Polyhedral characterization of discrete dynamic programming. *Operations research*, 38(1):127–138.

Slav Petrov, Aria Haghighi, and Dan Klein. 2008. Coarse-to-fine syntactic machine translation using language projections. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 108–116, Honolulu, Hawaii, October. Association for Computational Linguistics.

Sebastian Riedel and James Clarke. 2006. Incremental integer linear programming for non-projective dependency parsing. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, EMNLP '06, pages 129–137, Stroudsburg, PA, USA. Association for Computational Linguistics.

Alexander M Rush, David Sontag, Michael Collins, and Tommi Jaakkola. 2010. On dual decomposition and linear programming relaxations for natural language processing. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1–11, Cambridge, MA, October. Association for Computational Linguistics.

Libin Shen, Jinxi Xu, and Ralph Weischedel. 2008. A new string-to-dependency machine translation algorithm with a target dependency language model. In *Proceedings of ACL-08: HLT*, pages 577–585, Columbus, Ohio, June. Association for Computational Linguistics.

D.A. Smith and J. Eisner. 2008. Dependency parsing by belief propagation. In *Proc. EMNLP*, pages 145–156.

D. Sontag, T. Meltzer, A. Globerson, T. Jaakkola, and Y. Weiss. 2008. Tightening LP relaxations for MAP using message passing. In *Proc. UAI*.

Roy W. Tromble and Jason Eisner. 2006. A fast finite-state relaxation method for enforcing global constraints on sequence decoding. In *Proceedings of*

the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, HLT-NAACL '06, pages 423–430, Stroudsburg, PA, USA. Association for Computational Linguistics.

M. Wainwright, T. Jaakkola, and A. Willsky. 2005. MAP estimation via agreement on trees: message-passing and linear programming. In *IEEE Transactions on Information Theory*, volume 51, pages 3697–3717.

Taro Watanabe, Hajime Tsukada, and Hideki Isozaki. 2006. Left-to-right target generation for hierarchical phrase-based translation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, ACL-44, pages 777–784, Morristown, NJ, USA. Association for Computational Linguistics.