Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL), Philadelphia, July 2002, pp. 279-286.

Dynamic programming for parsing and estimation of stochastic unification-based grammars[*]

Stuart Geman
Division of Applied Mathematics
Brown University
geman@dam.brown.edu

Mark Johnson
Cognitive and Linguistic Sciences
Brown University
Mark_Johnson@Brown.edu

Abstract

Stochastic unification-based grammars (SUBGs) define exponential distributions over the parses generated by a unification-based grammar (UBG). Existing algorithms for parsing and estimation require the enumeration of all of the parses of a string in order to determine the most likely one, or in order to calculate the statistics needed to estimate a grammar from a training corpus. This paper describes a graph-based dynamic programming algorithm for calculating these statistics from the packed UBG parse representations of Maxwell and Kaplan (1995) which does not require enumerating all parses. Like many graphical algorithms, the dynamic programming algorithm's complexity is worst-case exponential, but is often polynomial. The key observation is that by using Maxwell and Kaplan packed representations, the required statistics can be rewritten as either the max or the sum of a product of functions. This is exactly the kind of problem which can be solved by dynamic programming over graphical models.

1 Introduction

Stochastic Unification-Based Grammars (SUBGs) use log-linear models (also known as exponential or MaxEnt models and Markov Random Fields) to define probability distributions over the parses of a unification grammar. These grammars can incorporate virtually all kinds of linguistically important constraints (including non-local and non-context-free constraints), and are equipped with a statistically sound framework for estimation and learning.

Abney (1997) pointed out that the non-context-free dependencies of a unification grammar require stochastic models more general than Probabilistic Context-Free Grammars (PCFGs) and Markov Branching Processes, and proposed the use of log-linear models for defining probability distributions over the parses of a unification grammar. Unfortunately, the maximum likelihood estimator Abney proposed for SUBGs seems computationally intractable since it requires statistics that depend on the set of all parses of all strings generated by the grammar. This set is infinite (so exhaustive enumeration is impossible) and presumably has a very complex structure (so sampling estimates might take an extremely long time to converge).

Johnson et al. (1999) observed that parsing and related tasks only require conditional distributions over parses given strings, and that such conditional distributions are considerably easier to estimate than joint distributions of strings and their parses. The conditional maximum likelihood estimator proposed by Johnson et al. requires statistics that depend on the set of all parses of the strings in the training cor-

[*] We would like to thank Eugene Charniak, Miyao Yusuke, Mark Steedman as well as Stefan Riezler and the team at PARC; naturally all errors remain our own. This research was supported by NSF awards DMS 0074276 and ITR IIS 0085940.

pus. For most linguistically realistic grammars this set is finite, and for moderate sized grammars and training corpora this estimation procedure is quite feasible.

However, our recent experiments involve training from the Wall Street Journal Penn Tree-bank, and repeatedly enumerating the parses of its 50,000 sentences is quite time-consuming. Matters are only made worse because we have moved some of the constraints in the grammar from the unification component to the stochastic component. This broadens the coverage of the grammar, but at the expense of massively expanding the number of possible parses of each sentence.

In the mid-1990s unification-based parsers were developed that do not enumerate all parses of a string but instead manipulate and return a "packed" representation of the set of parses. This paper describes how to find the most probable parse and the statistics required for estimating a SUBG from the packed parse set representations proposed by Maxwell III and Kaplan (1995). This makes it possible to avoid explicitly enumerating the parses of the strings in the training corpus.

The methods proposed here are analogues of the well-known dynamic programming algorithms for Probabilistic Context-Free Grammars (PCFGs); specifically the Viterbi algorithm for finding the most probable parse of a string, and the Inside-Outside algorithm for estimating a PCFG from unparsed training data.[1] In fact, because Maxwell and Kaplan packed representations are just Truth Maintenance System (TMS) representations (Forbus and de Kleer, 1993), the statistical techniques described here should extend to non-linguistic applications of TMSs as well.

Dynamic programming techniques have been applied to log-linear models before. Lafferty et al. (2001) mention that dynamic programming can be used to compute the statistics required for conditional estimation of log-linear models based on context-free grammars where the properties can include arbitrary functions of the input string. Miyao and Tsujii (2002) (which appeared after this paper was accepted) is the closest related work we know of. They describe a technique for calculating the statistics required to estimate a log-linear parsing model with non-local properties from packed feature forests.

[1]However, because we use conditional estimation, also known as discriminative training, we require at least some discriminating information about the correct parse of a string in order to estimate a stochastic unification grammar.

The rest of this paper is structured as follows. The next section describes unification grammars and Maxwell and Kaplan packed representation. The following section reviews stochastic unification grammars (Abney, 1997) and the statistical quantities required for efficiently estimating such grammars from parsed training data (Johnson et al., 1999). The final substantive section of this paper shows how these quantities can be defined directly in terms of the Maxwell and Kaplan packed representations.

The notation used in this paper is as follows. Variables are written in upper case italic, e.g., X, Y, etc., the sets they range over are written in script, e.g., $\mathcal{X}, \mathcal{Y}$, etc., while specific values are written in lower case italic, e.g., x, y, etc. In the case of vector-valued entities, subscripts indicate particular components.

2 Maxwell and Kaplan packed representations

This section characterises the properties of unification grammars and the Maxwell and Kaplan packed parse representations that will be important for what follows. This characterisation omits many details about unification grammars and the algorithm by which the packed representations are actually constructed; see Maxwell III and Kaplan (1995) for details.

A *parse* generated by a unification grammar is a *finite* subset of a set $\mathcal{F}$ of features. Features are parse fragments, e.g., chart edges or arcs from attribute-value structures, out of which the packed representations are constructed. For this paper it does not matter exactly what features are, but they are intended to be the atomic entities manipulated by a dynamic programming parsing algorithm. A grammar defines a set Ω of well-formed or grammatical parses. Each parse $\omega \in \Omega$ is associated with a string of words $Y(\omega)$ called its *yield*. Note that except for trivial grammars $\mathcal{F}$ and Ω are infinite.

If y is a string, then let $\Omega(y) = \{\omega \in \Omega | Y(\omega) = y\}$ and $\mathcal{F}(y) = \bigcup_{\omega \in \Omega(y)} \{f \in \omega\}$. That is, $\Omega(y)$ is the set of parses of a string y and $\mathcal{F}(y)$ is the set of features appearing in the parses of y. In the grammars of interest here $\Omega(y)$ and hence also $\mathcal{F}(y)$ are finite.

Maxwell and Kaplan's packed representations often provide a more compact representation of the set of parses of a sentence than would be obtained by merely listing each parse separately. The intuition behind these packed representations is that for most strings y, many of the features in $\mathcal{F}(y)$ occur in many of the parses $\Omega(y)$. This is often the case in natural language, since the same substructure can appear as a component of many different parses.

Packed feature representations are defined in terms of conditions on the values assigned to a vector of variables X. These variables have no direct linguistic interpretation; rather, each different assignment of values to these variables identifies a set of features which constitutes one of the parses in the packed representation. A *condition* a on X is a function from $\mathcal{X}$ to $\{0, 1\}$. While for uniformity we write conditions as functions on the entire vector X, in practice Maxwell and Kaplan's approach produces conditions whose value depends only on a few of the variables in X, and the efficiency of the algorithms described here depends on this.

A *packed representation* of a finite set of parses is a quadruple $R = (\mathcal{F}', X, N, \alpha)$, where:

- $\mathcal{F}' \supseteq \mathcal{F}(y)$ is a finite set of features,
- X is a finite vector of *variables*, where each variable X_ℓ ranges over the finite set $\mathcal{X}_\ell$,
- N is a finite set of conditions on X called the *no-goods*,[2] and
- α is a function that maps each feature $f \in \mathcal{F}'$ to a condition α_f on X.

A vector of values x *satisfies the no-goods* N iff $N(x) = 1$, where $N(x) = \prod_{\eta \in N} \eta(x)$. Each x that satisfies the no-goods *identifies* a parse $\omega(x) = \{f \in \mathcal{F}' | \alpha_f(x) = 1\}$, i.e., ω is the set of features whose conditions are satisfied by x. We require that each parse be identified by a *unique* value satisfying the no-goods. That is, we require that:

$$\forall x, x' \in \mathcal{X} \text{ if } N(x) = N(x') = 1 \text{ and } \omega(x) = \omega(x') \text{ then } x = x' \quad (1)$$

Finally, a packed representation R *represents* the set of parses $\Omega(R)$ that are identified by values that satisfy the no-goods, i.e., $\Omega(R) = \{\omega(x) | x \in \mathcal{X}, N(x) = 1\}$.

Maxwell III and Kaplan (1995) describes a parsing algorithm for unification-based grammars that takes as input a string y and returns a packed representation R such that $\Omega(R) = \Omega(y)$, i.e., R represents the set of parses of the string y. The SUBG parsing and estimation algorithms described in this paper use Maxwell and Kaplan's parsing algorithm as a subroutine.

3 Stochastic Unification-Based Grammars

This section reviews the probabilistic framework used in SUBGs, and describes the statistics that must be calculated in order to estimate the parameters of a SUBG from parsed training data. For a more detailed exposition and descriptions of regularization and other important details, see Johnson et al. (1999).

The probability distribution over parses is defined in terms of a finite vector $g = (g_1, \ldots, g_m)$ of properties. A *property* is a real-valued function of parses Ω. Johnson et al. (1999) placed no restrictions on what functions could be properties, permitting properties to encode arbitrary global information about a parse. However, the dynamic programming algorithms presented here require the information encoded in properties to be local with respect to the features $\mathcal{F}$ used in the packed parse representation. Specifically, we require that properties be defined on features rather than parses, i.e., each feature $f \in \mathcal{F}$ is associated with a finite vector of real values $(g_1(f), \ldots, g_m(f))$ which define the property functions for parses as follows:

$$g_k(\omega) = \sum_{f \in \omega} g_k(f), \text{ for } k = 1 \ldots m. \quad (2)$$

That is, the property values of a parse are the sum of the property values of its features. In the usual case, some features will be associated with a single property (i.e., $g_k(f)$ is equal to 1 for a specific value

[2]The name "no-good" comes from the TMS literature, and was used by Maxwell and Kaplan. However, here the no-goods actually identify the *good* variable assignments.

of k and 0 otherwise), and other features will be associated with no properties at all (i.e., $g(f) = 0$).

This requires properties be very local with respect to features, which means that we give up the ability to define properties arbitrarily. Note however that we can still encode essentially arbitrary linguistic information in properties by adding specialised features to the underlying unification grammar. For example, suppose we want a property that indicates whether the parse contains a reduced relative clauses headed by a past participle (such "garden path" constructions are grammatical but often almost incomprehensible, and alternative parses not including such constructions would probably be preferred). Under the current definition of properties, we can introduce such a property by modifying the underlying unification grammar to produce a certain "diacritic" feature in a parse just in case the parse actually contains the appropriate reduced relative construction. Thus, while properties are required to be local relative to features, we can use the ability of the underlying unification grammar to encode essentially arbitrary non-local information in features to introduce properties that also encode non-local information.

A Stochastic Unification-Based Grammar is a triple (U, g, θ), where U is a unification grammar that defines a set Ω of parses as described above, $g = (g_1, \ldots, g_m)$ is a vector of property functions as just described, and $\theta = (\theta_1, \ldots, \theta_m)$ is a vector of non-negative real-valued parameters called *property weights*. The probability $\mathrm{P}_\theta(\omega)$ of a parse $\omega \in \Omega$ is:

$$\begin{aligned} \mathrm{P}_\theta(\omega) &= \frac{W_\theta(\omega)}{Z_\theta}, \text{ where:} \\ W_\theta(\omega) &= \prod_{j=1}^{m} \theta_j^{g_j(\omega)}, \text{ and} \\ Z_\theta &= \sum_{\omega' \in \Omega} W_\theta(\omega') \end{aligned}$$

Intuitively, if $g_j(\omega)$ is the number of times that property j occurs in ω then θ_j is the 'weight' or 'cost' of each occurrence of property j and Z_θ is a normalising constant that ensures that the probability of all parses sums to 1.

Now we discuss the calculation of several important quantities for SUBGs. In each case we show that the quantity can be expressed as the value that maximises a product of functions or else as the sum of a product of functions, each of which depends on a small subset of the variables X. These are the kinds of quantities for which dynamic programming graphical model algorithms have been developed.

3.1 The most probable parse

In parsing applications it is important to be able to extract the most probable (or MAP) parse $\hat{\omega}(y)$ of string y with respect to a SUBG. This parse is:

$$\hat{\omega}(y) \quad = \quad \operatorname*{argmax}_{\omega \in \Omega(y)} W_\theta(\omega)$$

Given a packed representation $(\mathcal{F}', X, N, \alpha)$ for the parses $\Omega(y)$, let $\hat{x}(y)$ be the x that identifies $\hat{\omega}(y)$. Since $W_\theta(\hat{\omega}(y)) > 0$, it can be shown that:

$$\begin{aligned} \hat{x}(y) &= \operatorname*{argmax}_{x \in \mathcal{X}} N(x) \prod_{j=1}^{m} \theta_j^{g_j(\omega(x))} \\ &= \operatorname*{argmax}_{x \in \mathcal{X}} N(x) \prod_{j=1}^{m} \theta_j^{\sum_{f \in \omega(x)} g_j(f)} \\ &= \operatorname*{argmax}_{x \in \mathcal{X}} N(x) \prod_{j=1}^{m} \theta_j^{\sum_{f \in \mathcal{F}'} \alpha_f(x) g_j(f)} \\ &= \operatorname*{argmax}_{x \in \mathcal{X}} N(x) \prod_{j=1}^{m} \prod_{f \in \mathcal{F}'} \theta_j^{\alpha_f(x) g_j(f)} \\ &= \operatorname*{argmax}_{x \in \mathcal{X}} N(x) \prod_{f \in \mathcal{F}'} \left(\prod_{j=1}^{m} \theta_j^{g_j(f)} \right)^{\alpha_f(x)} \\ &= \operatorname*{argmax}_{x \in \mathcal{X}} \prod_{\eta \in N} \eta(x) \prod_{f \in \mathcal{F}'} h_{\theta,f}(x) \qquad (3) \end{aligned}$$

where $h_{\theta,f}(x) = \prod_{j=1}^{m} \theta_j^{g_j(f)}$ if $\alpha_f(x) = 1$ and $h_{\theta,f}(x) = 1$ if $\alpha_f(x) = 0$. Note that $h_{\theta,f}(x)$ depends on exactly the same variables in X as α_f does. As (3) makes clear, finding $\hat{x}(y)$ involves maximising a product of functions where each function depends on a subset of the variables X. As explained below, this is exactly the kind of maximisation that can be solved using graphical model techniques.

3.2 Conditional likelihood

We now turn to the estimation of the property weights θ from a training corpus of parsed data $D = (\omega_1, \ldots, \omega_n)$. As explained in Johnson et al. (1999), one way to do this is to find the θ that maximises the conditional likelihood of the training corpus parses given their yields. (Johnson et al. actually maximise conditional likelihood regularized with a Gaussian prior, but for simplicity we ignore this here). If y_i is the yield of the parse ω_i, the conditional likelihood of the parses given their yields is:

$$L_D(\theta) = \prod_{i=1}^{n} \frac{W_\theta(\omega_i)}{Z_\theta(\Omega(y_i))}$$

where $\Omega(y)$ is the set of parses with yield y and:

$$Z_\theta(S) = \sum_{\omega \in S} W_\theta(\omega).$$

Then the maximum conditional likelihood estimate $\hat{\theta}$ of θ is $\hat{\theta} = \mathrm{argmax}_\theta\, L_D(\theta)$.

Now calculating $W_\theta(\omega_i)$ poses no computational problems, but since $\Omega(y_i)$ (the set of parses for y_i) can be large, calculating $Z_\theta(\Omega(y_i))$ by enumerating each $\omega \in \Omega(y_i)$ can be computationally expensive.

However, there is an alternative method for calculating $Z_\theta(\Omega(y_i))$ that does not involve this enumeration. As noted above, for each yield $y_i, i = 1, \ldots, n$, Maxwell's parsing algorithm returns a packed feature structure R_i that represents the parses of y_i, i.e., $\Omega(y_i) = \Omega(R_i)$. A derivation parallel to the one for (3) shows that for $R = (\mathcal{F}', X, N, \alpha)$:

$$Z_\theta(\Omega(R)) = \sum_{x \in \mathcal{X}} \prod_{\eta \in N} \eta(x) \prod_{f \in \mathcal{F}'} h_{\theta,f}(x) \quad (4)$$

(This derivation relies on the isomorphism between parses and variable assignments in (1)). It turns out that this type of sum can also be calculated using graphical model techniques.

3.3 Conditional Expectations

In general, iterative numerical procedures are required to find the property weights θ that maximise the conditional likelihood $L_D(\theta)$. While there are a number of different techniques that can be used, all of the efficient techniques require the calculation of conditional expectations $E_\theta[g_k|y_i]$ for each property g_k and each sentence y_i in the training corpus, where:

$$\begin{aligned} E_\theta[g|y] &= \sum_{\omega \in \Omega(y)} g(\omega) \mathrm{P}_\theta(\omega|y) \\ &= \frac{\sum_{\omega \in \Omega(y)} g(\omega) W_\theta(\omega)}{Z_\theta(\Omega(y))} \end{aligned}$$

For example, the Conjugate Gradient algorithm, which was used by Johnson et al., requires the calculation not just of $L_D(\theta)$ but also its derivatives $\partial L_D(\theta)/\partial \theta_k$. It is straight-forward to show:

$$\frac{\partial L_D(\theta)}{\partial \theta_k} = \frac{L_D(\theta)}{\theta_k} \sum_{i=1}^{n} \left(g_k(\omega_i) - E_\theta[g_k|y_i]\right).$$

We have just described the calculation of $L_D(\theta)$, so if we can calculate $E_\theta[g_k|y_i]$ then we can calculate the partial derivatives required by the Conjugate Gradient algorithm as well.

Again, let $R = (\mathcal{F}', X, N, \alpha)$ be a packed representation such that $\Omega(R) = \Omega(y_i)$. First, note that (2) implies that:

$$E_\theta[g_k|y_i] = \sum_{f \in \mathcal{F}'} g_k(f)\, \mathrm{P}(\{\omega : f \in \omega\}|y_i).$$

Note that $\mathrm{P}(\{\omega : f \in \omega\}|y_i)$ involves the sum of weights over all $x \in \mathcal{X}$ subject to the conditions that $N(x) = 1$ and $\alpha_f(x) = 1$. Thus $\mathrm{P}(\{\omega : f \in \omega\}|y_i)$ can also be expressed in a form that is easy to evaluate using graphical techniques.

$$\begin{aligned} & Z_\theta(\Omega(R)) \mathrm{P}_\theta(\{\omega : f \in \omega\}|y_i) \\ = & \sum_{x \in \mathcal{X}} \alpha_f(x) \prod_{\eta \in N} \eta(x) \prod_{f' \in \mathcal{F}'} h_{\theta,f'}(x) \quad (5) \end{aligned}$$

4 Graphical model calculations

In this section we briefly review graphical model algorithms for maximising and summing products of functions of the kind presented above. It turns out that the algorithm for maximisation is a generalisation of the Viterbi algorithm for HMMs, and the algorithm for computing the summation in (5) is a generalisation of the forward-backward algorithm for HMMs (Smyth et al., 1997). Viewed abstractly, these algorithms simplify these expressions by moving common factors over the max or sum operators respectively. These techniques are now relatively standard; the most well-known approach involves junction trees (Pearl, 1988; Cowell, 1999). We adopt the approach approach described by Geman and Kochanek (2000), which is a straightforward generalization of HMM dynamic programming with minimal assumptions and programming overhead. However, in principle any of the graphical model computational algorithms can be used.

The quantities (3), (4) and (5) involve maximisation or summation over a product of functions, each of which depends only on the values of a subset of the variables X. There are dynamic programming algorithms for calculating all of these quantities, but for reasons of space we only describe an algorithm for finding the maximum value of a product of functions. These graph algorithms are rather involved. It may be easier to follow if one reads Example 1 before or in parallel with the definitions below.

To explain the algorithm we use the following notation. If x and x' are both vectors of length m then $x =_j x'$ iff x and x' disagree on at most their jth components, i.e., $x_k = x'_k$ for $k = 1, \ldots, j-1, j+1, \ldots m$. If f is a function whose domain is $\mathcal{X}$, we say that f *depends on* the set of variables $d(f) = \{X_j | \exists x, x' \in \mathcal{X}, x =_j x', f(x) \neq f(x')\}$. That is, $X_j \in d(f)$ iff changing the value of X_j can change the value of f.

The algorithm relies on the fact that the variables in $X = (X_1, \ldots, X_n)$ are ordered (e.g., X_1 precedes X_2, etc.), and while the algorithm is correct for any variable ordering, its efficiency may vary dramatically depending on the ordering as described below. Let $\mathcal{H}$ be any set of functions whose domains are X. We partition $\mathcal{H}$ into disjoint subsets $\mathcal{H}_1, \ldots, \mathcal{H}_{n+1}$, where $\mathcal{H}_j$ is the subset of $\mathcal{H}$ that depend on X_j but do not depend on any variables ordered before X_j, and $\mathcal{H}_{n+1}$ is the subset of $\mathcal{H}$ that do not depend on any variables at all (i.e., they are constants).[3] That is, $\mathcal{H}_j = \{H \in \mathcal{H} | X_j \in d(H), \forall i < j \; X_i \notin d(H)\}$ and $\mathcal{H}_{n+1} = \{H \in \mathcal{H} | d(H) = \emptyset\}$.

As explained in section 3.1, there is a set of functions $\mathcal{A}$ such that the quantities we need to calculate have the general form:

$$M_{\max} = \max_{x \in \mathcal{X}} \prod_{A \in \mathcal{A}} A(x) \quad (6)$$

$$\hat{x} = \operatorname*{argmax}_{x \in \mathcal{X}} \prod_{A \in \mathcal{A}} A(x). \quad (7)$$

$M_{\max}$ is the maximum value of the product expression while $\hat{x}$ is the value of the variables at which the maximum occurs. In a SUBG parsing application $\hat{x}$ identifies the MAP parse.

[3]Strictly speaking this does not necessarily define a partition, as some of the subsets $\mathcal{H}_j$ may be empty.

The procedure depends on two sequences of functions $M_i, i = 1, \ldots, n+1$ and $V_i, i = 1, \ldots, n$. Informally, M_i is the maximum value attained by the subset of the functions $\mathcal{A}$ that depend on one of the variables $X_1, \ldots, X_i$, and V_i gives information about the value of X_i at which this maximum is attained.

To simplify notation we write these functions as functions of the entire set of variables X, but usually depend on a much smaller set of variables. The M_i are real valued, while each V_i ranges over $\mathcal{X}_i$. Let $\mathcal{M} = \{M_1, \ldots, M_n\}$. Recall that the sets of functions $\mathcal{A}$ and $\mathcal{M}$ can be both be partitioned into disjoint subsets $A_1, \ldots, \mathcal{A}_{n+1}$ and $\mathcal{M}_1, \ldots, \mathcal{M}_{n+1}$ respectively on the basis of the variables each A_i and M_i depend on. The definition of the M_i and $V_i, i = 1, \ldots, n$ is as follows:

$$M_i(x) = \max_{\substack{x' \in \mathcal{X} \\ \text{s.t. } x' =_i x}} \prod_{A \in \mathcal{A}_i} A(x') \prod_{M \in \mathcal{M}_i} M(x') \quad (8)$$

$$V_i(x) = \operatorname*{argmax}_{\substack{x' \in \mathcal{X} \\ \text{s.t. } x' =_i x}} \prod_{A \in \mathcal{A}_i} A(x') \prod_{M \in \mathcal{M}_i} M(x')$$

M_{n+1} receives a special definition, since there is no variable X_{n+1}.

$$M_{n+1} = \left(\prod_{A \in \mathcal{A}_{n+1}} A \right) \left(\prod_{M \in \mathcal{M}_{n+1}} M \right) \quad (9)$$

The definition of M_i in (8) may look circular (since M appears in the right-hand side), but in fact it is not. First, note that M_i depends only on variables ordered after X_i, so if $M_j \in \mathcal{M}_i$ then $j < i$. More specifically,

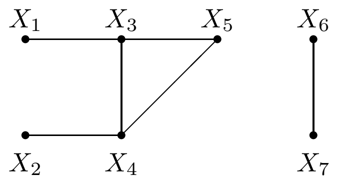
$$d(M_i) = \left(\bigcup_{A \in \mathcal{A}_i} d(A) \cup \bigcup_{M \in \mathcal{M}_i} d(M) \right) \setminus \{X_i\}.$$

Thus we can compute the M_i in the order $M_1, \ldots, M_{n+1}$, inserting M_i into the appropriate set $\mathcal{M}_k$, where $k > i$, when M_i is computed.

We claim that $M_{\max} = M_{n+1}$. (Note that M_{n+1} and M_n are constants, since there are no variables ordered after X_n). To see this, consider the tree $\mathcal{T}$ whose nodes are the M_i, and which has a directed edge from M_i to M_j iff $M_i \in \mathcal{M}_j$ (i.e., M_i appears in the right hand side of the definition (8) of M_j). $\mathcal{T}$ has a unique root M_{n+1}, so there is a path from every M_i to M_{n+1}. Let $i \prec j$ iff there is a path from M_i to M_j in this tree. Then a simple induction shows that M_j is a function from $d(M_j)$ to a maximisation over each of the variables X_i where $i \prec j$ of $\prod_{i \prec j, A \in \mathcal{A}_i} A$.

Further, it is straightforward to show that $V_i(\hat{x}) = \hat{x}_i$ (the value $\hat{x}$ assigns to X_i). By the same arguments as above, $d(V_i)$ only contains variables ordered after X_i, so $V_n = \hat{x}_n$. Thus we can evaluate the V_i in the order $V_n, \ldots, V_1$ to find the maximising assignment $\hat{x}$.

Example 1 *Let $X = \{X_1, X_2, X_3, X_4, X_5, X_6, X_7\}$ and set $\mathcal{A} = \{a(X_1, X_3), b(X_2, X_4), c(X_3, X_4, X_5), d(X_4, X_5), e(X_6, X_7)\}$. We can represent the sharing of variables in $\mathcal{A}$ by means of a undirected graph $\mathcal{G}_\mathcal{A}$, where the nodes of $\mathcal{G}_\mathcal{A}$ are the variables X and there is an edge in $\mathcal{G}_\mathcal{A}$ connecting X_i to X_j iff $\exists A \in \mathcal{A}$ such that both $X_i, X_j \in d(A)$. $\mathcal{G}_\mathcal{A}$ is depicted below.*



Starting with the variable X_1, we compute M_1 and V_1:

$$\begin{aligned} M_1(x_3) &= \max_{x_1 \in \mathcal{X}_1} a(x_1, x_3) \\ V_1(x_3) &= \operatorname*{argmax}_{x_1 \in \mathcal{X}_1} a(x_1, x_3) \end{aligned}$$

We now proceed to the variable X_2.

$$\begin{aligned} M_2(x_4) &= \max_{x_2 \in \mathcal{X}_2} b(x_2, x_4) \\ V_2(x_4) &= \operatorname*{argmax}_{x_2 \in \mathcal{X}_2} b(x_2, x_4) \end{aligned}$$

Since M_1 belongs to $\mathcal{M}_3$, it appears in the definition of M_3.

$$\begin{aligned} M_3(x_4, x_5) &= \max_{x_3 \in \mathcal{X}_3} c(x_3, x_4, x_5) M_1(x_3) \\ V_3(x_4, x_5) &= \operatorname*{argmax}_{x_3 \in \mathcal{X}_3} c(x_3, x_4, x_5) M_1(x_3) \end{aligned}$$

Similarly, M_4 is defined in terms of M_2 and M_3.

$$\begin{aligned} M_4(x_5) &= \max_{x_4 \in \mathcal{X}_4} d(x_4, x_5) M_2(x_4) M_3(x_4, x_5) \\ V_4(x_5) &= \operatorname*{argmax}_{x_4 \in \mathcal{X}_4} d(x_4, x_5) M_2(x_4) M_3(x_4, x_5) \end{aligned}$$

Note that M_5 is a constant, reflecting the fact that in $\mathcal{G}_\mathcal{A}$ the node X_5 is not connected to any node ordered after it.

$$\begin{aligned} M_5 &= \max_{x_5 \in \mathcal{X}_5} M_4(x_5) \\ V_5 &= \operatorname*{argmax}_{x_5 \in \mathcal{X}_5} M_4(x_5) \end{aligned}$$

The second component is defined in the same way:

$$\begin{aligned} M_6(x_7) &= \max_{x_6 \in \mathcal{X}_6} e(x_6, x_7) \\ V_6(x_7) &= \operatorname*{argmax}_{x_6 \in \mathcal{X}_6} e(x_6, x_7) \\ M_7 &= \max_{x_7 \in \mathcal{X}_7} M_6(x_7) \\ V_7 &= \operatorname*{argmax}_{x_7 \in \mathcal{X}_7} M_6(x_7) \end{aligned}$$

The maximum value for the product $M_8 = M_{\max}$ is defined in terms of M_5 and M_7.

$$M_{\max} = M_8 = M_5 M_7$$

Finally, we evaluate $V_7, \ldots, V_1$ to find the maximising assignment $\hat{x}$.

$$\begin{aligned} \hat{x}_7 &= V_7 \\ \hat{x}_6 &= V_6(\hat{x}_7) \\ \hat{x}_5 &= V_5 \\ \hat{x}_4 &= V_4(\hat{x}_5) \\ \hat{x}_3 &= V_3(\hat{x}_4, \hat{x}_5) \\ \hat{x}_2 &= V_2(\hat{x}_4) \\ \hat{x}_1 &= V_1(\hat{x}_3) \end{aligned}$$

We now briefly consider the computational complexity of this process. Clearly, the number of steps required to compute each M_i is a polynomial of order $|d(M_i)| + 1$, since we need to enumerate all possible values for the argument variables $d(M_i)$ and for each of these, maximise over the set $\mathcal{X}_i$. Further, it is easy to show that in terms of the graph $\mathcal{G}_\mathcal{A}$, $d(M_j)$ consists of those variables $X_k, k > j$ reachable by a path starting at X_j and all of whose nodes except the last are variables that precede X_j.

Since computational effort is bounded above by a polynomial of order $|d(M_i)| + 1$, we seek a variable ordering that bounds the maximum value of $|d(M_i)|$. Unfortunately, finding the ordering that minimises the maximum value of $|d(M_i)|$ is an NP-complete problem. However, there are several efficient heuristics that are reputed in graphical models community to produce good visitation schedules. It may be that they will perform well in the SUBG parsing applications as well.

5 Conclusion

This paper shows how to apply dynamic programming methods developed for graphical models to SUBGs to find the most probable parse and to obtain the statistics needed for estimation directly from Maxwell and Kaplan packed parse representations. i.e., without expanding these into individual parses. The algorithm rests on the observation that so long as features are local to the parse fragments used in the packed representations, the statistics required for parsing and estimation are the kinds of quantities that dynamic programming algorithms for graphical models can perform. Since neither Maxwell and Kaplan's packed parsing algorithm nor the procedures described here depend on the details of the underlying linguistic theory, the approach should apply to virtually any kind of underlying grammar.

Obviously, an empirical evaluation of the algorithms described here would be extremely useful. The algorithms described here are exact, but because we are working with unification grammars and apparently arbitrary graphical models we cannot polynomially bound their computational complexity. However, it seems reasonable to expect that if the linguistic dependencies in a sentence typically factorize into largely non-interacting cliques then the dynamic programming methods may offer dramatic computational savings compared to current methods that enumerate all possible parses.

It might be interesting to compare these dynamic programming algorithms with a standard unification-based parser using a best-first search heuristic. (To our knowledge such an approach has not yet been explored, but it seems straightforward: the figure of merit could simply be the sum of the weights of the properties of each partial parse's fragments). Because such parsers prune the search space they cannot guarantee correct results, unlike the algorithms proposed here. Such a best-first parser might be accurate when parsing with a trained grammar, but its results may be poor at the beginning of parameter weight estimation when the parameter weight estimates are themselves inaccurate.

Finally, it would be extremely interesting to compare these dynamic programming algorithms to the ones described by Miyao and Tsujii (2002). It seems that the Maxwell and Kaplan packed representation may permit more compact representations than the disjunctive representations used by Miyao et al., but this does not imply that the algorithms proposed here are more efficient. Further theoretical and empirical investigation is required.

References

Steven Abney. 1997. Stochastic Attribute-Value Grammars. *Computational Linguistics*, 23(4):597–617.

Robert Cowell. 1999. Introduction to inference for Bayesian networks. In Michael Jordan, editor, *Learning in Graphical Models*, pages 9–26. The MIT Press, Cambridge, Massachusetts.

Kenneth D. Forbus and Johan de Kleer. 1993. *Building problem solvers*. The MIT Press, Cambridge, Massachusetts.

Stuart Geman and Kevin Kochanek. 2000. Dynamic programming and the representation of soft-decodable codes. Technical report, Division of Applied Mathematics, Brown University.

Mark Johnson, Stuart Geman, Stephen Canon, Zhiyi Chi, and Stefan Riezler. 1999. Estimators for stochastic "unification-based" grammars. In *The Proceedings of the 37th Annual Conference of the Association for Computational Linguistics*, pages 535–541, San Francisco. Morgan Kaufmann.

John Lafferty, Andrew McCallum, and Fernando Pereira. 2001. Conditional Random Fields: Probabilistic models for segmenting and labeling sequence data. In *Machine Learning: Proceedings of the Eighteenth International Conference (ICML 2001)*, Stanford, California.

John T. Maxwell III and Ronald M. Kaplan. 1995. A method for disjunctive constraint satisfaction. In Mary Dalrymple, Ronald M. Kaplan, John T. Maxwell III, and Annie Zaenen, editors, *Formal Issues in Lexical-Functional Grammar*, number 47 in CSLI Lecture Notes Series, chapter 14, pages 381–481. CSLI Publications.

Yusuke Miyao and Jun'ichi Tsujii. 2002. Maximum entropy estimation for feature forests. In *Proceedings of Human Language Technology Conference 2002*, March.

Judea Pearl. 1988. *Probabalistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, California.

Padhraic Smyth, David Heckerman, and Michael Jordan. 1997. Probabilistic Independence Networks for Hidden Markov Models. *Neural Computation*, 9(2):227–269.