Technische Universität Dresden
Fakultät Informatik
Institut für Theoretische Informatik
Lehrstuhl Grundlagen der Programmierung

Diplomarbeit

k-Best A* Parsing: Implementation and Heuristics

Johannes Osterholzer

Eingereicht am 1. Dezember 2011
Bearbeitet vom 1. Juli 2011
bis zum 30. November 2011

Verantwortlicher Hochschullehrer:
Prof. Dr.-Ing. habil. Heiko Vogler
Betreuer:
Dr. Torsten Stüber, M. Sc.

Contents

1 Introduction

The scientific field of Natural Language Processing (NLP) deals with capturing natural, i.e. human, languages in a form that enables machine-based processing and translation. In this field, there has been a long tradition of using *phrase-based* formalisms in order to do so, which account for the hierarchical structure of natural sentences, whose basic building blocks—subject, predicate, object, etc.—are each again recursively built up from smaller units, down to the level of single words of the sentence. An analysis of the phrasal structure of a sentence can then be represented by an unranked tree, often called the *(full-)parse tree* of the sentence. In fact, it can be shown that phrase-based models are necessary to grasp the complexity of human language, compare the argument that English is not a regular language, made by Chomsky (1957).

Phrase-Based Models

The most noted phrase-based models are *context-free grammars* (Chomsky, 1956, originally called phrase-structure grammars), but there are also several possible alternatives. One of these are *tree adjoining grammars* (introduced by Joshi, Levy, and Takahashi, 1975) and, as a restriction of this formalism, *tree insertion grammars* (introduced as *lexicalized context-free grammars* by Schabes and Waters, 1994). The elementary objects manipulated by such grammars, and the products of their derivations, are trees instead of mere symbols. This suits the modelling of the phrasal structure of languages quite well, giving focus to parse trees instead of flat sentences.

Tree adjoining grammars allow the basic operations of substitution and adjoining. Both of those have straightforward applications for natural languages. The languages generated by such tree-adjoining grammars are a proper superset of the set of context-free languages (cf. Joshi and Schabes, 1991), while tree-insertion grammars generate only context-free languages (see Schabes and Waters, 1994).

Probabilistic Grammars

As detailed by Manning and Schütze (1999, ch. 1), there are convincing arguments against a *categorical* view of natural language, i.e., against the belief that one can give a fixed grammar for, e.g., modern English, which generates all grammatically correct *(grammatical)* sentences of this language. This view is problematic because of the ambiguity inherent to spoken (and written) language: there are examples for constructions which one speaker might deem as correct, but might be accepted by the next one only with a furrowed brow, or might not be accepted at all. Moreover, since language constantly develops, there are phrases which may have been usual several decades ago, but may sound rather odd or incorrect to the modern ear. Nevertheless, it would be hard to find a sharp date up to which such a construction was considered grammatical, and after which it was not used any longer. Instead, it gradually faded out of use.

These observations lead to the notion that a theory of natural language should not so much try to divide all human utterances into two disjoint sets of grammatical and ungrammatical sentences, but rather concern itself with how *usual* such a sentence is. A measure of this can be derived with *probabilistic* measures, and, indeed, there are probabilistic versions of most grammar formalisms which assign each of their production rules a certain probability. The probability of a derivation is then defined as the product of the probabilities of the rules used in it. In Section 4.1 of this work, we will apply this idea to one of the grammars mentioned above and give a formal definition of *probabilistic tree insertion grammars.*

The KA* Algorithm

In such a probabilistic framework, given a grammar, the problem arises to generate its possible derivations, in the order of their respective probabilities. As the set of these derivations is frequently infinite and its computation may be prohibitively expensive, one often settles with determining a number k of the best, i.e. most probable derivations. One efficient solution for this *k-best problem* is the *KA* algorithm* (Pauls and Klein, 2009a), which uses a supplied external heuristic function, as well as internal heuristics generated on-the-fly, to prioritize its search over the derivation space of a grammar.

In Chapter 3, we detail the idea behind an optimized *lazy* variant of this algorithm and present a Haskell implementation written during the course of this work. In this implementation, KA* does not operate directly on the supplied grammar, but instead on a *probabilistic hypergraph* representation, whose derivations are equivalent to those of the grammar. This allows decoupling between the concrete type of the supplied grammar, be it a context-free grammar, a tree insertion grammar, or another formalism, and the KA* algorithm. In a way, one can think of this hypergraph representation as an *interface* provided to KA*. Of course, for this to work out, the transformation from grammars to hypergraphs has to be defined. In our case, for probabilistic tree insertion grammars, this is accomplished along with their definition in Section 4.1.

State-Split and the Learning of Grammars

Probabilistic grammars bring the additional advantage that there are algorithms for their unsupervised learning. Supplied with a corpus, i.e., in this case, with a collection of hand-parsed natural language sentences, those algorithms generate the probabilistic grammar that is most likely to generate the analyses contained in that corpus, or at least a good candidate for such a grammar. Their performance in capturing human language can then be evaluated on another corpus, giving rise to several metrics which may be used to compare different grammars (for a basic introduction, refer to Charniak, 1997).

Section 4.2 contains a description of one particular method of learning probabilistic grammars, the *State-Split* method (first presented by Petrov, Barrett, Thibaux, and Klein, 2006). State-Split starts out with a corpus of analyses of sentences, together with a grammar grasping this corpus in some way. This grasp is then continually refined by splitting the grammar's phrasal categories into two by the use of *annotations*, reestimating the new production probabilities with the help of the *Expectation-Maximization (EM)* algorithm, which facilitates maximum-likelihood estimation from incomplete data, and merging back together subcategories which

turn out to be irrelevant for achieving higher likelihood. Again, in our case, we will define State-Split not directly on the grammars involved, but instead on their hypergraph representations, promoting abstraction from their concrete types.

Subsection 4.2.3 will detail a definition for the EM algorithm, applied to annotated derivations, with less annotated derivations serving as incomplete data. This formulation of EM is based on the work by Prescher (2005). One should take note that there are much more general definitions for EM, which is applicable for many problems where a maximum-likelihood estimate from incomplete data is sought. Examples for such problems are already given in the algorithm's introductory paper by Dempster, Laird, and Rubin (1977).

Subsequently, Subsection 4.2.4 addresses the *Inside-Outside* algorithm, which can be seen as a dynamic programming instance of EM. Inside-Outside was originally presented for corpora of unparsed sentences by Baker (1979), and adapted to corpora of parse trees by Pereira and Schabes (1992). We will show how, in this case, the Inside-Outside algorithm emerges from EM quite naturally, by reshaping a few formulas.

Discussion: Induction of Heuristics

Finally, we intended to describe a method, presented by Pauls and Klein (2009b), to extract admissible and consistent heuristics, applicable to the KA* algorithm, from the intermediate products of the State-Split procedure. However, as the assertion this method relies on does not hold, at least for our case, we had to give up on this intent.

Instead, in Chapter 5 we will try to outline the idea of Pauls and Klein and the conjecture it rests on, as well as a small counterexample to the latter. We will proceed by discussing possible reasons for this surprising discrepancy.

2 Prelude

Note that in the following we will assume $\mathbb{N}$ to be the set of non-negative integers, i.e.

$$\mathbb{N} = \{0, 1, 2, \ldots\}.$$

To avoid tedious repetition, we introduce the abbreviation $[n] = \{1, \ldots, n\}$ for every $n \in \mathbb{N}$. Note that $[0] = \emptyset$. Moreover, $\mathbb{R}$ will denote the set of the real numbers, and

$$\mathbb{R}_{\geq 0} = \{r \in \mathbb{R} \mid r \geq 0\}$$

the set of the non-negative reals. Presuming a function $f : A \to B$, for every subset $S \subseteq B$, we define its *preimage* under f as

$$f^{-1}(S) = \{a \in A \mid f(a) \in S\}.$$

When we have finite sets, we will sometimes abbreviate $f^{-1}(\{s_1, \ldots, s_n\})$ by $f^{-1}\{s_1, \ldots, s_n\}$. For the same function $f : A \to B$, and for every subset $S \subseteq A$, its *image* under f is given by

$$f(S) = \{f(a) \mid a \in S\}.$$

The *Kronecker delta*[1] is a notational shorthand which allows to condition equations on the equality of two terms. Assume a set A and two values $a, b \in A$, then

$$\delta(a, b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise.} \end{cases}$$

The *powerset* 2^A of a set A is declared as the set of all subsets of A:

$$2^A = \{B \mid B \subseteq A\}.$$

Finally, the set of *words* of length $n \in \mathbb{N}$ over a set A is defined as

$$A^n = \{a_1 \cdots a_n \mid a_1, \ldots, a_n \in A\},$$

where for $n = 0$ the *empty word* is denoted by ε. The set of words over A is correspondingly defined as

$$A^* = \bigcup_{n \in \mathbb{N}} A^n.$$

If we have a word $w = a_1 \cdots a_n \in A^n$ and an index $i \in [n]$, we will define access to the i-th symbol by $w(i) = a_i$.

[1] Traditionally denoted by δ_{ab}, but since we will deal with comparatively long terms in place of a and b, we will write $\delta(a, b)$ instead.

2.1 Algebraic Necessities

Since later we will have to give a canonical sequence for elements in certain finite sets, we must define a few basic concepts regarding partial and total orders.

Definition 2.1. A *partially ordered set (poset)* is a structure $\mathbf{P} = (P, \leq)$ where P is a set and $\leq$ a binary relation on P such that for every $p, q, r \in P$

- $p \leq p$ *(reflexivity)*,
- $p \leq q$ and $q \leq p$ implies $p = q$ *(antisymmetry)* and
- $p \leq q$ and $q \leq r$ implies $p \leq r$ *(transitivity)*.

For such a poset $\mathbf{P}$, we also say that its carrier set P is partially ordered by $\leq$. If we additionally have $p \leq q$ or $q \leq p$ for every $p, q \in P$ *(totality)*, we call $\mathbf{P}$ a *totally ordered set* resp. P totally ordered by $\leq$. We will write $p < q$ as a shorthand for "$p \leq q$ and $p \neq q$".

Definition 2.2. If a set A is partially ordered by $\leq$, we can define a poset $(A^*, \leq_*)$ where A^* is the set of words over A, as defined above, and where for $v, w \in A^*$ we have $v \leq_* w$ if

- v is a prefix of w or
- there are $u, v', w' \in A^*$, $a, b \in A$ such that $v = uav'$, $w = ubw'$ and $a < b$.

We call the defined partial order the *lexicographic order* on A^*. If A is totally ordered, A^* is also totally ordered (cf. Baader and Nipkow, 1998)

Definition 2.3. A *monoid* is an algebraic structure $\mathbf{M} = \langle M, \circ, e \rangle$ comprising a binary operation $\circ\colon M \times M \to M$ and an element $e \in M$ such that

- the operation $\circ$ is *associative:* $a \circ (b \circ c) = (a \circ b) \circ c$ for every $a, b, c \in M$ and
- e acts as a *neutral element* regarding this operation: $a \circ e = e \circ a = a$ for every $a \in M$.

We call $\mathbf{M}$ *commutative* if $a \circ b = b \circ a$ for every $a, b \in M$ and *cancellative* if we can cancel equations from the left and the right, i.e. if $a \circ b = a \circ c$ implies $b = c$, and $a \circ c = b \circ c$ implies $a = b$, for every $a, b, c \in M$.

A partial order $(M, \leq)$ is said to be *compatible* to a monoid $\mathbf{M} = \langle M, \circ, e \rangle$ if $a \leq b$ implies $a \circ c \leq b \circ c$ and $c \circ a \leq c \circ b$ for every $a, b, c \in M$. Supplied with such a compatible order, $\mathbf{M}$ is called an *ordered monoid* or *ordered by* $\leq$. If $(M, \leq)$ is even a total order, we call $\mathbf{M}$ *totally ordered by* $\leq$.

Example 2.4. The most prominent monoid in the following developments will be the monoid of *probabilities,* $\mathbf{Pr} = \langle [0, 1], \cdot, 1 \rangle$, with $[a, b]$ as the interval $\{r \in \mathbb{R} \mid a \leq r \leq b\}$, for $a, b \in \mathbb{R}$, and $\cdot$ as multiplication on the real numbers. Note that this monoid is as well commutative as cancellative, and ordered by the well-known order $\leq$ on the real numbers.

2.2 Probability Theory

We also need a few basic definitions from the field of probability theory. Since in the work at hand, we will not have to concern ourselves with the paradoxa often arising with uncountably infinite sets, most of the measure-theoretic underpinnings of modern probability will be done away with in the following. For a thorough mathematical treatise of this field, refer to Kallenberg (2001) instead. Manning and Schütze (1999, ch. 2) also give a short introduction to probability theory as it is utilized in Natural Language Processing.

Definition 2.5. Assuming a countable set Ω, called the *sample space,* we call a function $p : \Omega \to [0, 1]$ a *probability mass function (pmf),* if

$$\sum_{\omega \in \Omega} p(\omega) = 1.$$

Such a pmf naturally induces a *probability distribution* $P : 2^{\Omega} \to [0, 1]$ *on* Ω,[2] defined by

$$P(A) = \sum_{a \in A} p(a).$$

A tuple (Ω, P) of a sample space Ω and a probability distribution P on Ω will henceforth be called a *probability space,* while the subsets $A \subseteq \Sigma$ will be designated as *events.*

Random elements give us the means to distill information from a probability space and to derive corresponding probabilities.

Definition 2.6. Let (Ω, P) be a probability space and S a countable set. A function $X : \Omega \to S$ is then called a *random element in* S. We can infer a probability distribution on S from X and P by defining

$$P(X \in B) = P(X^{-1}(B))$$

for every $B \subseteq S$. If we have $B = \{b\}$, we will also write $P(X = b)$ instead of $P(X \in B)$. Similarly, for $k > 1$ random elements $X_i : \Omega \to S_i$, $i \in [k]$, we define

$$P(X_1 \in B_1, \ldots, X_k \in B_k) = P(X_1^{-1}(B_1) \cap \cdots \cap X_k^{-1}(B_k)).$$

Given a function $f : S \to T$, with S and T countable, and $X : S \to T$ a random element, we can derive the random element $f(X)$, which should not be interpreted as application of the function f to X, but instead as an alternative for denoting $f \circ X$.[3] A random element $X : \Omega \to S$ will be called a *random variable* if $S = \mathbb{R}$.[4]

[2] In the literature, this distribution is mostly defined on a *σ-algebra*, i.e., a set system on Ω with certain properties. As mentioned above, we will omit such intricacies. Refer to Prescher (2005) for a justification of this omission.

[3] Of course, this notation is meant to resemble application of f to X. This is justified by the fact that, in many cases, it is intuitive to think of X not as a function, but as a mere element from the codomain of X, hence the name random *element.*

[4] In many works covering probability theory this distinction is not made and random *elements* are referred to as random *variables* generally. However, as this naming issue only seems to be a matter of different tastes, we will stick for now with the term *random element*, following Kallenberg (2001).

Definition 2.7. Given a sample space Ω, a *probability model on* Ω is a non-empty set $\mathscr{M}$ of probability distributions on Ω. The *unrestricted probability model* $\mathscr{M}(\Omega)$ is defined by

$$\mathscr{M}(\Omega) = \left\{ P : 2^{\Omega} \to [0,1] \mid P \text{ is a probability distribution on } \Omega \right\}.$$

All other probability models $\mathscr{N}$ on Ω (i.e. all probability models $\mathscr{N}$ with $\mathscr{N} \neq \mathscr{M}(\Omega)$) are called *restricted*.

Definition 2.8. For a probability space (Ω, P), the *conditional probability* $P(A \mid B)$ denotes the *posterior* probability that some event $A \subseteq \Omega$ occurs given the knowledge that $B \subseteq \Omega$ occurs certainly or has already occured. It can be computed as

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

if $P(B) \neq 0$. $P(B)$ is then referred to as the *prior* probability. Note that this definition extends naturally to random elements: Assume, e.g., two random elements $X : \Omega \to S$ and $Y : \Omega \to T$, with S and T countable sets, then

$$P(X \in A \mid Y \in B) = \frac{P(X \in A, Y \in B)}{P(Y \in B)} = \frac{P(X^{-1}(A) \cap Y^{-1}(B))}{P(Y^{-1}(B))}.$$

For certain sums of probability distributions over two random elements, one can factor out one of these random elements, arriving at the *marginal distribution*.

Lemma 2.9. *For a probability space* (Ω, P) *and two random elements* $X : \Omega \to S$, $Y : \Omega \to T$, *if we know the joint probability distribution*

$$P(X = s, Y = t)$$

for every $s \in S$ *and* $t \in T$, *we can derive the* marginal probability distributions

$$P(X = s) = \sum_{t \in T} P(X = s, Y = t)$$

as well as

$$P(Y = t) = \sum_{s \in S} P(X = s, Y = t). \qquad (\square)$$

Next, let us introduce the idea of a *maximum-likelihood estimate (mle),* defined as an element of a probability model which maximizes the *likelihood* on a certain *corpus* containing statistical data. So first of all, we have to define the notion of corpora:

Definition 2.10. For a sample space Ω, we call a function $f : \Omega \to \mathbb{R}$ a *corpus* if for every $x \in \Omega$ we have $f(x) \geq 0$. For a corpus, we also call the elements $x \in \Omega$ *types* and their values $f(x)$ the corresponding *type frequencies*. The *size* $|f|$ of the corpus is given by

$$|f| = \sum_{x \in \Omega} f(x)$$

and we call f *non-empty* if $|f| > 0$ as well as *finite* if $|f| \in \mathbb{R}$.

This allows us to denote the *likelihood* of a corpus, giving us a measure of how much a probability distribution "fits" the statistical distribution of the underlying corpus.

Definition 2.11. Assume a sample space Ω, a probability model $\mathscr{M}$ on Ω and a non-empty, finite corpus $f : \Omega \to [0, 1]$. The *likelihood* (or just *probability*) of f allocated by an element of the probability model $P \in \mathscr{M}$ is defined as

$$L(f; P) = \prod_{x \in \Omega} P(x)^{f(x)},$$

where we define $0^0 = 1$.[5] We call an assumed probability distribution $\hat{P}$ a *maximum-likelihood estimate (mle) of $\mathscr{M}$ on f* if

$$L(f; \hat{P}) \in \arg\max_{P \in \mathscr{M}} L(f; P),$$

i.e., if the likelihood it allocates to f is maximal given the supposed probability model.

2.3 Hypergraphs and Probabilities

The concept of (directed) graphs, which comprise a set of nodes and a set of edges that connect pairs of nodes, is well-known in the fields of Mathematics and Computer Science. Weighted graphs are often used as data structures for path-finding problems, as e.g. in the A* algorithm. One can generalize the idea of graphs by allowing edges to connect more than two nodes, this leads to the notion of *hypergraphs*. In the work at hand, we will only concern ourselves with *directed* hypergraphs—where every hyperedge connects a set of *tail nodes* to a set of *head nodes*—which are also *functional,* i.e., each hyperedge has exactly one head node.

Definition 2.12. A *hypergraph* is a tuple $G = (N, E, \mu, g)$ where

- N is called the set of *nodes* of the hypergraph,
- E the set of *hyperedges,*
- $\mu : E \to N^* \times N$ assigns a number of *tail nodes* and a unique *head node* to every edge,
- $g \in N$ is a certain node in the hypergraph, called the *goal node.*

We call G *finite* if both sets N and E are finite. For every edge $e \in E$, where $\mu(e) = (b_1 \cdots b_k, a)$, we denote its tail nodes by $\mathrm{tl}(e) = b_1 \cdots b_k$, its head node by $\mathrm{hd}(e) = a$ and its *arity* by $\mathrm{ar}(e) = |\mathrm{tl}(e)|$.

As for traditional graphs, we can equip every hyperedge of a hypergraph with a weight over some monoid. In the general case, we will even assign each edge a weight function, of the same arity as the hyperedge.

Definition 2.13. Presuming a monoid $\mathbf{M} = \langle M, \circ, e \rangle$, an $\mathbf{M}$*-weighted hypergraph* is a tuple $G = (N, E, \mu, g, w)$ where

[5] This definition is strongly suggested by the two well-known facts that the cardinality of the function space A^A on a set A is $|A|^{|A|}$ and that there is exactly one function from the empty set onto itself.

- (N, E, μ, g) is a hypergraph as defined in Def. 2.12 and
- $w = (w_e : M^{\mathrm{ar}(e)} \to M \mid e \in E)$ is a family of functions, supplying every edge with an *edge weight function.*

If for every $e \in E$ there is an $m \in M$ such that the corresponding edge weight function is of the form $w_e(m_1, \ldots, m_n) = m_1 \circ \cdots \circ m_n \circ m$ for $n = \mathrm{ar}(e)$ and every $m_1, \ldots, m_n \in M$, we say that G has *multiplicative weights* and write $w(e) = m$ as a shorthand.

Definition 2.14. A *probabilistic hypergraph (phg)* is the same as a **Pr**-weighted hypergraph with multiplicative weights. For such graphs, we will write p instead of w for the edge weight function and call $p(e) = w(e)$ the *edge probability* of an edge $e \in E$. The class of all probabilistic hypergraphs will be abbreviated by PHG. A phg G is referred to as *proper* if, for every $a \in N$,

$$\sum_{\substack{e \in E \\ \mathrm{hd}(e) = a}} p(e) = 1.$$

Now we can define the notion of derivations in hypergraphs, which can be seen as analogous to the concept of *paths* in a graph, but lifted to the world of hypergraphs. They can be understood as trees, or as (typed) terms, over the set of hyperedges.

Definition 2.15. Presuming a weighted hypergraph $G = (N, E, \mu, g, w)$, we define the set D^a_G of *derivations of a in G* for every node $a \in N$ to be the smallest set D with

$$D = \{e(d_1, \ldots, d_k) \mid e \in E, \mu(e) = (b_1 \cdots b_k, a), d_i \in D^{b_i}_G \text{ for } i \in [k]\}.$$

We can then denote the set of *derivations in G* by

$$D_G = \bigcup_{a \in N} D^a_G.$$

For a derivation $d = e(d_1, \ldots, d_k)$, we define the shorthand notation $\mathrm{hd}(d) = \mathrm{hd}(e)$, i.e., a derivation's head is defined as the head of the hyperedge at its root.

Contexts can be thought of as derivations with a "gap", symbolized by $\square$. This gap can act as destination for substituting in other contexts or derivations.

Definition 2.16. For a hypergraph $G = (N, E, \mu, g)$, we demand that the symbol $\square \notin E$. Then, for every node $a, b \in N$, we define the set $C^a_{G,b}$ of *a-contexts for b* as the smallest set C with

- $\square \in C$ and
- $\{c\,[e(d_1, \ldots, d_{i-1}, \square, d_{i+1}, \ldots, d_n)] \mid c \in C^a_{b_0, G}, d_j \in D^{b_j}_G \text{ for } j \in [n] \setminus \{i\}\} \subseteq C$ for every $e \in E$, $\mu(e) = (b_1 \cdots b_n, b_0)$, $i \in [n]$ with $b_i = b$.

Here, $c[c']$ denotes the substitution of all leaf nodes $\square$ in c with c'. As for derivations, we let

$$C^a_G = \bigcup_{b \in N} C^a_{G,b}$$

denote the set of all a-contexts.

For derivations in a weighted hypergraph, we can determine their weight by inductive application of the weight functions (resp., multiplication of the weights) of the hyperedges contained in the derivation. In the case of hypergraphs with multiplicative weights, this technique works also for contexts (with non-multiplicative weights, we would have to represent context-weights by unary operations on the monoid).

Definition 2.17. We can compute the weight of a derivation $d = e(d_1, \ldots, d_k) \in D_G$ in a $\langle M, \circ, e\rangle$-weighted hypergraph $G = (N, E, \mu, g, w)$ by structural induction:

$$\text{weight}_G(d) = w_e(\text{weight}_G(d_1), \ldots, \text{weight}_G(d_k)).$$

We call a phg G *consistent* if

$$\sum_{d \in D_G^g} \text{weight}_G(d) = 1.$$

In the case that G is consistent and proper, weight_G, restricted to the domain D_G^g, is a pmf. Then we will write $P(\cdot \mid G)$ for the probability distribution on D_G^g induced by weight_G, compare Def. 2.5.[6]

> *In the course of this work, whenever we write about a probability distribution $P(\cdot \mid G)$ induced by a phg $G = (N, E, \mu, g, p)$, we implicitly assume G to be proper and consistent (else the notation would not make much sense). Moreover, we will sometimes also denote probabilities $P(d \mid G)$ of derivations d of nodes that are not the goal node g. Then we read this as $P(d \mid G')$ with $G' = (N, E, \mu, a, p)$ instead and again assume G' is proper and consistent.*

Definition 2.18. The weight of a context $c \in C_G^a$ in a $\langle M, \circ, e\rangle$-weighted hypergraph $G = (N, E, \mu, g, w)$ with multiplicative weights can be defined by structural induction as follows:

$$\text{weight}_G(\Box) = e$$

as well as

$$\begin{aligned}&\text{weight}_G(c[e(d_1, \ldots, d_{i-1}, \Box, d_{i+1}, \ldots d_k)])\\ &\quad = \text{weight}_G(c) \circ w_e \circ \text{weight}_G(d_1) \circ \cdots \circ \text{weight}_G(d_{i-1})\\ &\qquad \circ \text{weight}_G(d_{i+1}) \circ \cdots \circ \text{weight}_G(d_k).\end{aligned}$$

Given a node $a \in N$ in a phg $G = (N, E, \mu, g, p)$, we can define its *Viterbi inside* and *outside score* $\beta^*(a)$ resp. $\alpha^*(a)$ as

$$\begin{aligned}\beta^*(a) &= \max\{\text{weight}_G(d) \mid d \in D_G^a\}\\ \alpha^*(a) &= \max\{\text{weight}_G(c) \mid c \in C_{G,a}^g\}\end{aligned}$$

[6] This way of denoting probability ditributions induced by a phg may seem like misuse of the notation for conditional probabilities, but it is intuitive (read $P(d \mid G)$ as "the probability of d under the knowledge of G's induced probability distribution") and consistent with the notation by, e.g., Manning and Schütze (1999) and Lari and Young (1991).

We call a derivation of (resp. g-context for) $a \in N$ with weight $\beta(a)$ (resp. $\alpha(a)$) a *best*, *lightest* or *Viterbi* derivation (context).

If we have a hypergraph which describes the possible structure of derivations, this allows us to define a probability model, comprising all probability distributions induced by the assumed hypergraph extended to a proper and consistent phg.

Definition 2.19. Let $G = (N, E, \mu, g)$ be an arbitrary hypergraph. The *probability model of* G (on $\Omega = D_G^g$) arises as

$$\mathscr{M}_G = \left\{ P(\cdot \mid G') \in \mathscr{M}(D_G^g) \mid p \in [0,1]^E,\ G' = (N, E, \mu, g, p) \text{ is proper and consistent} \right\}.$$

2.4 Unranked Trees

In the study of tree automata, one usually deals with *ranked* trees, which can be also thought of as (typed) terms over some ranked alphabet of underlying symbols, each with a unique arity. But with natural languages and the trees denoting their phrase structure, dealing with ranked symbols becomes cumbersome. For example, in one linguistic role, an NP (noun phrase) part-of-speech tag might have several children, in another one only one. This leads us to the notion of *unranked trees*.

Definition 2.20. Assuming two disjoint set Σ and A, we can define the set of *unranked trees over* Σ *with leaves also from* A, $U_\Sigma(A)$, as the smallest set fulfilling the following properties:

- For every $a \in A$ we have that $a \in U_\Sigma(A)$.
- For every $n \in \mathbb{N}$ and $\sigma \in \Sigma$, if $\xi_1, \ldots, \xi_n \in U_\Sigma(A)$, then also $\sigma(\xi_1, \ldots, \xi_n) \in U_\Sigma(A)$.

Similar to ranked trees, we can now define various functions on $U_\Sigma(A)$.

Definition 2.21. First of all, we give a function generating the set of *positions* in an unranked tree. A position uniquely determines an element in an unranked tree.

$$\begin{aligned}
&\mathrm{pos}\colon U_\Sigma(A) \to 2^{\mathbb{N}^*} \\
&\mathrm{pos}(a) = \{\varepsilon\} && \text{for } a \in A \\
&\mathrm{pos}(\sigma(\xi_1, \ldots, \xi_n)) = \{\varepsilon\} \cup \bigcup_{i \in [n]} \{i w \mid w \in \mathrm{pos}(\xi_i)\} && \text{for } \sigma \in \Sigma
\end{aligned}$$

We can also restrict this function to only return the tree positions labeled with elements from a certain set $\Delta \subseteq \Sigma \cup A$:

$$\begin{aligned}
&\mathrm{pos}_\Delta\colon U_\Sigma(A) \to 2^{\mathbb{N}^*} \\
&\mathrm{pos}_\Delta(a) = \{\varepsilon\} && \text{for } a \in A \cap \Delta \\
&\mathrm{pos}_\Delta(a) = \emptyset && \text{for } a \in A \setminus \Delta \\
&\mathrm{pos}_\Delta(\sigma(\xi_1, \ldots, \xi_n)) = \{\varepsilon\} \cup \bigcup_{i \in [n]} \{i w \mid w \in \mathrm{pos}(\xi_i)\} && \text{for } \sigma \in \Sigma \cap \Delta \\
&\mathrm{pos}_\Delta(\sigma(\xi_1, \ldots, \xi_n)) = \bigcup_{i \in [n]} \{i w \mid w \in \mathrm{pos}(\xi_i)\} && \text{for } \sigma \in \Sigma \setminus \Delta
\end{aligned}$$

Definition 2.22. Moreover, we have a function computing the positions of the *leaves* of an unranked tree, i.e. of the tree nodes with no children.

$$\begin{aligned}
&\mathrm{lv}\colon U_\Sigma(A) \to 2^{\mathbb{N}^*} \\
&\mathrm{lv}(a) = \{\varepsilon\} && \text{for } a \in A \\
&\mathrm{lv}(\alpha()) = \{\varepsilon\} && \text{for } \alpha \in \Sigma \\
&\mathrm{lv}(\sigma(\xi_1, \ldots, \xi_n)) = \bigcup_{i \in [n]} \{i\,w \mid w \in \mathrm{lv}(\xi_i)\} && \text{for } n \geq 1,\ \sigma \in \Sigma
\end{aligned}$$

It is easy to see that $\mathrm{lv}(\xi) \subseteq \mathrm{pos}(\xi)$ for every $\xi \in U_\Sigma(A)$. Therefore, for every such ξ, we can define the positions of its *non-leaves*, i.e. of its nodes with children, by $\mathrm{nlv}(\xi) = \mathrm{pos}(\xi) \setminus \mathrm{lv}(\xi)$. Again, we can adapt these functions to return only positions labeled with elements from a set $\Delta \subseteq \Sigma \cup A$: for every $\xi \in U_\Sigma(A)$, let $\mathrm{lv}_\Delta(\xi) = \mathrm{lv}(\xi) \cap \mathrm{pos}_\Delta(\xi)$ and $\mathrm{nlv}_\Delta(\xi) = \mathrm{nlv}(\xi) \cap \mathrm{pos}_\Delta(\xi)$.

Definition 2.23. For every set $\Delta \subseteq \Sigma \cup A$, the function yield_Δ returns the left-to-right sequence of the leaf nodes from Δ in an unranked tree and is defined by

$$\begin{aligned}
&\mathrm{yield}_\Delta\colon U_\Sigma(A) \to \Delta^* \\
&\mathrm{yield}_\Delta(a) = \varepsilon && \text{for } a \in A \setminus \Delta \\
&\mathrm{yield}_\Delta(a) = a && \text{for } a \in A \cap \Delta \\
&\mathrm{yield}_\Delta(\alpha()) = \varepsilon && \text{for } \alpha \in \Sigma \setminus \Delta \\
&\mathrm{yield}_\Delta(\alpha()) = \alpha && \text{for } \alpha \in \Sigma \cap \Delta \\
&\mathrm{yield}_\Delta(\sigma(\xi_1, \ldots, \xi_n)) = \mathrm{yield}_\Delta(\xi_1) \cdots \mathrm{yield}_\Delta(\xi_n) && \text{for } n \geq 1,\ \sigma \in \Sigma
\end{aligned}$$

Definition 2.24. Similarly, for every set $\Delta \subseteq \Sigma \cup A$, every unranked tree $\xi \in U_\Sigma(A)$ and one of its leaf positions $w \in \mathrm{lv}(\xi)$, $\mathrm{yield}'_\Delta(\xi, w)$ computes the sequence $\mathrm{yield}_\Delta(\xi)$ without the label for the position w.

$$\begin{aligned}
&\mathrm{yield}'_\Delta\colon U_\Sigma(A) \times \mathbb{N}^* \to \Delta^* \\
&\mathrm{yield}'_\Delta(a, w) = \varepsilon && \text{for } a \in A \text{ with } a \notin \Delta \text{ or } w = \varepsilon \\
&\mathrm{yield}'_\Delta(a, w) = a && \text{for } a \in A \cap \Delta \text{ and } w \neq \varepsilon \\
&\mathrm{yield}'_\Delta(\alpha(), w) = \varepsilon && \text{for } \alpha \in \Sigma \text{ with } \alpha \notin \Delta \text{ or } w = \varepsilon \\
&\mathrm{yield}'_\Delta(\alpha(), w) = \alpha && \text{for } \alpha \in \Sigma \cap \Delta \text{ and } w \neq \varepsilon \\
&\mathrm{yield}'_\Delta(\sigma(\xi_1, \ldots, \xi_n), i\,w) \\
&\quad = \mathrm{yield}_\Delta(\xi_1) \cdots \mathrm{yield}_\Delta(\xi_{i-1})\, \mathrm{yield}'_\Delta(\xi_i, w)\, \mathrm{yield}_\Delta(\xi_{i+1}) \cdots \mathrm{yield}_\Delta(\xi_n) \\
&\qquad && \text{for } n \geq 1,\ \sigma \in \Sigma
\end{aligned}$$

Definition 2.25. Given a tree $\xi = \sigma(\xi_1, \ldots, \xi_n) \in U_\Sigma(A)$ and one of its positions $w \in \mathrm{pos}(\xi)$, we can give its label $\xi(w)$ at the position w by recursively defining

$$\begin{aligned}
&\xi(\varepsilon) = \sigma \\
&\xi(i\,w) = \xi_i(w).
\end{aligned}$$

Similarly, its subtree ξ_w at the position w is given by

$$\begin{aligned}\xi|_\varepsilon &= \xi\\ \xi|_{iw} &= \xi_i|_w.\end{aligned}$$

Definition 2.26. Given two trees $\xi, \zeta \in U_\Sigma(A)$ and a position $w \in \mathrm{pos}(\xi)$, we define the tree obtained by replacing ξ's subtree at w by ζ as $\xi[\zeta]_w$ according to

$$\begin{aligned}\xi[\zeta]_\varepsilon &= \zeta\\ \sigma(\xi_1,\dots,\xi_k)[\zeta]_{iw} &= \sigma(\xi_1,\dots,\xi_{i-1},\xi_i[\zeta]_w,\xi_{i+1},\dots,\xi_k).\end{aligned}$$

Similarly, for every tree $\xi \in U_\Sigma(A)$, list of leaf positions $\omega = w_1 \cdots w_k \in \mathrm{lv}(\xi)^*$, $k \in \mathbb{N}$ with pairwise distinct w_i, $i \in [k]$, and other trees $\zeta_1,\dots,\zeta_k \in U_\Sigma(A)$, we can define *parallel substitution* as

$$\xi[\zeta_1,\dots,\zeta_k]_\omega = \left(\cdots\left((\xi[\zeta_1]_{w_1})[\zeta_2]_{w_2}\right)\cdots\right)[\zeta_k]_{w_k}$$

for $k > 0$, else

$$\xi[]_\varepsilon = \xi.$$

Note that the k substitutions in this definition do not interfere with each other since they were restricted to only occur at leaf positions.

3 Lazy KA*

In the following chapter we will start out with describing the principles behind the lazy variant of the KA* algorithm for heuristic-based search of k best derivations in a probabilistic hypergraph. This comprises a short recap of the adaptation of the algorithm's standard version, detailed for pcfgs by Pauls and Klein (2009a), to probabilistic hypergraphs.

Subsequently, we will give a short argument how the probabilistic monoid's monotony properties can be exploited in order to cut back on the generation of unnecessary assignments, and show how the standard algorithm has to be modified to achieve this.

The following section will then detail the realization of the developed ideas and describe an implementation of Lazy KA* in the non-strict functional programming language Haskell (Peyton-Jones, 2003), together with an account of some problems which arose during its development process and certain optimizations which were applied.

3.1 Recapping KA*

Supplied with a probabilistic hypergraph G and a corresponding external heuristic function h, the KA* algorithm computes the k best derivations of the graph's goal node in G, often referred to as the *k-best problem* or just *k-best*. Let us first define the mathematical notion of such a heuristic.

Definition 3.1. For a phg $G = (N, E, \mu, g, p)$, we call a function $h : N \to [0,1]$ a *heuristic function* for G. Under the premise that for every edge $e \in E$ with $\mu(e) = (b_1 \cdots b_n, a)$, and derivations $d_j \in D_G^{b_j}$ for $j \in [n]$,

$$P(d_i \mid G) \cdot h(d_i) \geq P(e(d_1, \ldots, d_n) \mid G) \cdot h(a)$$

for every $i \in [n]$, the heuristic h is said to be *consistent*. This property is also referred to as *monotony* of the heuristic. Consistency of heuristics also implies that they are *admissible*, i.e., that for every node $a \in N$,

$$h(a) \geq \alpha^*(a),$$

in particular $h(g) = 1$. We call an admissible heuristic h *perfect* when we even have $h(a) = \alpha^*(a)$ for every node $a \in N$.

Given a heuristic like this, KA* tackles the k-best problem by a prioritized search in the *search graph* G' generated from G and h. This KA* search graph's node set consists of three different types of *items*, called *inside items, outside items* and *derivation items*.

The supplied external heuristic function contributes to the priorities of inside items in the graph search. This allows KA* to efficiently compute the Viterbi inside scores of G's nodes with the computation of best derivations of inside items in G'. These inside scores are then

used as a heuristic for the determination of best derivations of outside items in G', providing the algorithm with the value of each node's Viterbi outside score. Finally, as clearly evident from Def. 3.1, the outside scores comprise a perfect heuristic in G, facilitating the efficient output of up to k best derivations, represented in G' by the derivation items. The KA* search graph can be defined as follows:

Definition 3.2. Let us assume a phg $G = (N, E, \mu, g, p)$ as well as a consistent and admissible heuristic function $h : N \to [0, 1]$. Then the *KA* search graph* derived from G and h is defined as $G' = (N', E', \mu', g', w')$, where

- the node set N' consists of three disjoint sets of *items*, respectively called *inside*, *outside* and *derivation* items:

$$N' = \{\mathrm{I}(a) \mid a \in N\} \cup \{\mathrm{O}(a) \mid a \in N\} \cup \{\mathrm{D}(d) \mid d \in D_G\}$$

  where $\mathrm{I}(\cdot)$, $\mathrm{O}(\cdot)$ etc. can be understood as mere syntactical constructs providing disjointness of the respective sets,

- the edge set E' is constructed from the original one according to

$$\begin{aligned} E' = &\{\mathrm{in}(e) \mid e \in E\} \cup \{\mathrm{switch}\} \cup \{\mathrm{out}(i, e) \mid e \in E, i \in [\mathrm{ar}(e)]\} \\ &\cup \{\mathrm{cat}(e, d_1, \ldots, d_n) \mid e \in E, \mu(e) = (b_1 \cdots b_n, a), d_i \in D_G^{b_i} \text{ for } i \in [n]\} \end{aligned}$$

  again with $\mathrm{in}(\cdot)$ etc. as syntactical "tags",

- the function $\mu' : E' \to N'^* \times N'$ can be defined by distinction on the supplied edge types, given $\mu(e) = (b_1 \cdots b_n, a)$:

$$\begin{aligned} \mu'(\mathrm{in}(e)) &= \big(\mathrm{I}(b_1) \cdots \mathrm{I}(b_n), \mathrm{I}(a)\big) \\ \mu'(\mathrm{switch}) &= (\mathrm{I}(g), \mathrm{O}(g)) \\ \mu'(\mathrm{out}(i, e)) &= \big(\mathrm{O}(a)\, \mathrm{I}(b_1) \cdots \mathrm{I}(b_n), \mathrm{O}(b_i)\big) \\ \mu'(\mathrm{cat}(e, d_1, \ldots, d_n)) &= \big(\mathrm{O}(a)\, \mathrm{D}(d_1) \cdots \mathrm{D}(d_n), \mathrm{D}\left(e(d_1, \ldots, d_n)\right)\big) \end{aligned}$$

- the goal node g' is some $\mathrm{D}(d)$ with $d \in D_G^g$,[1]

- and the weight function $w'_\sigma : [0, 1]^{\mathrm{ar}(\sigma)} \to [0, 1]$ for $\sigma \in E'$ given by

$$\begin{aligned} w'_{\mathrm{in}(e)}(w_1, \ldots, w_n) &= w_1 \cdots w_n \cdot p(e) \\ w'_{\mathrm{switch}}(w) &= 1 \\ w'_{\mathrm{out}(i,e)}(w_0, w_1, \ldots, w_n) &= w_0 \cdot w_1 \cdots w_{i-1} \cdot w_{i+1} \cdots w_n \cdot p(e) \\ w'_{\mathrm{cat}(e,d_1,\ldots,d_n)}(w_0, w_1, \ldots, w_n) &= w_1 \cdots w_n \cdot p(e). \end{aligned}$$

To alleviate the reading effort, in this chapter we will use greek letters when quantifying over nodes and derivations in the search graph G', while reserving latin ones for the original graph.

[1] Or left undefined, it will not play any role in the execution of the KA* algorithm.

Note that the hyperedges in G' can be interpreted as statements about *dependency* between different items. For example, an edge $\sigma \in E'$ with $\mu(\sigma) = (\mathrm{O}(a)\mathrm{I}(b_1)\mathrm{I}(b_2), \mathrm{O}(b_2))$ can be read as "in order to compute the Viterbi outside score for the node b_2, we already have to know the outside score of a and inside scores for b_1 and b_2." This interpretation will come in handy when we move on to Lazy KA*.

The *priority function* specifies the order in which the items of the search graph are explored:

Definition 3.3. The priority function $\psi = (\psi_\sigma \mid \sigma \in E')$ is a family of functions $\psi_\sigma : M^{\mathrm{ar}(\sigma)} \to M$ (for every $\sigma \in E'$) defined by

$$\begin{aligned}
\psi_{\mathrm{in}(e)}(w_1, \ldots, w_n) &= w_1 \cdots w_n \cdot w_e \cdot h(\mathrm{hd}(e)) \\
\psi_{\mathrm{switch}}(w) &= w \\
\psi_{\mathrm{out}(i,e)}(w_0, w_1, \ldots, w_n) &= w_0 \cdot w_1 \cdots w_n \cdot w_e \\
\psi_{\mathrm{cat}(e,d_1,\ldots,d_n)}(w_0, w_1, \ldots, w_n) &= w_0 \cdot w_1 \cdots w_n \cdot w_e.
\end{aligned}$$

Algorithm 3.1 KA* Search

Input: A phg $G = (N, E, \mu, g, w)$, a heuristic $h : N \to [0, 1]$ and a number $k \in \mathbb{N}$
Output: A list $L \in \left(D_G^g\right)^*$ of best derivations, with length $|L| \leq k$

```
Construct the search graph G' = (N', E', μ', g', w') acc. to Def. 3.2
S ← ∅, Q ← ∅, L ← ε
for all σ ∈ E' such that ar(σ) = 0 do
    insert (hd(σ) = w'_σ(), ψ_σ()) into Q
end for
while Q ≠ ∅ and |L| < k do
    pop a maximal priority assignment (Φ = p, ψ) from Q
    if there is no assignment (Φ = p') ∈ S then
        S ← S ∪ {(Φ = p)}
        if Φ = D(d) with hd(d) = g then
            append d to L
        end if
        for all σ ∈ E' do
            if for every i ∈ [ar(σ)] exists (Φ_i = p_i) ∈ S s.t. Φ_i = tl(σ)(i) and Φ_j = Φ for some
            j ∈ [ar(σ)] then
                insert (hd(σ) = w'_σ(p_1, ..., p_ar(σ)), ψ_σ(p_1, ..., p_ar(σ))) into Q
            end if
        end for
    end if
end while
```

The search graph G' is then traversed as detailed in Alg. 3.1. Throughout its execution, KA* maintains two basic data structures. The first one is called the *chart* S, a set containing items which have already been explored, together with their determined best probabilities. Such pairs of items $\Phi \in N'$ and probabilities $p \in [0, 1]$ are called *assignments* and written as $(\Phi = p)$, but this is just a fancy way of denoting the tuple (Φ, p).

The other data structure is called the *agenda*[2] Q and contains *prioritized assignments* (i.e. assignments together with some priority) which can be thought of as assignments on the chart *in spe,* in the following sense: in each iteration of the algorithm's main loop, a maximally prioritized assignment $(\Phi = w, \psi)$ is removed from the agenda and a check is performed whether another assignment of Φ is already contained in the chart. If this is not the case, $(\Phi = w)$ gets promoted to an assignment and is inserted into the chart. Subsequently, all new prioritized assignments derivable from $(\Phi = w)$ and other assignments already in the chart are generated and inserted into the agenda, possibly to be processed in one of the later iterations.

The algorithm exits the loop when there are no more prioritized assignments left on the agenda to process or when enough derivations of G's goal node have been generated. These constitute the algorithm's succeeding output.

Keep in mind that this traversal does not necessarily imply that inside, outside and derivation items are processed exactly in this order: on the contrary, it might well be the case that the exploration of, e.g., some outside item $\mathrm{O}(a)$ is triggered well before the chart is saturated with assignments to all possible inside items.

3.2 Laziness

Having refreshed their knowledge of KA*, one might wonder if there is not room for optimizations. Let us imagine an iteration of the algorithm's main loop, where some derivation item $\mathrm{D}(d_j)$ is removed from the agenda. Our next step would be to combine the derivation d_j with all other compatible derivations $d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_k$ on the chart, along every edge $e \in E$ coming into question, and to immediately generate all possible derivation items $\mathrm{D}(e(d_1, \ldots, d_k))$, storing them as prioritized assignments on the agenda. But that could amount to an awful lot of new assignments! Not only would their generation, their future removal from the agenda, as well as their possibly even unnecessary processing, cost computation time, they would also require precious memory, which could otherwise be used more sensibly for, e.g., another treebank holding linguistic data.

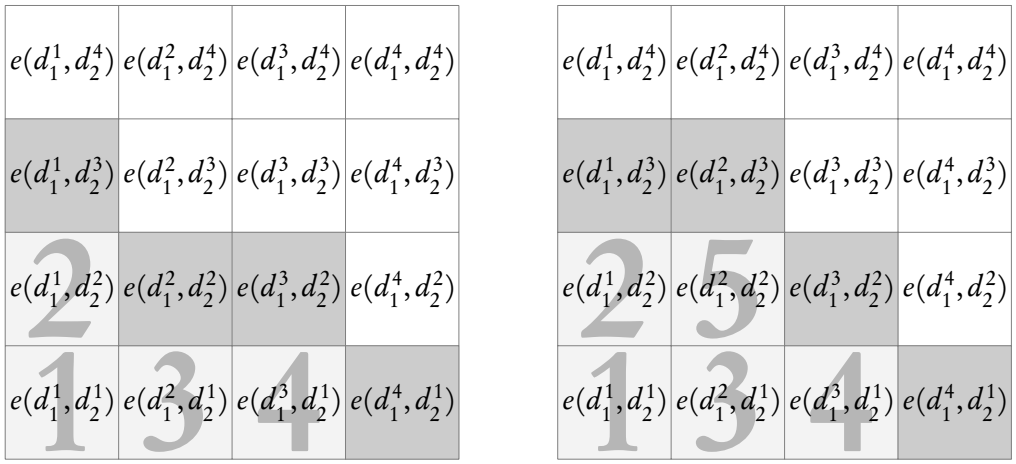
Providentially, we can devise a way to cut back on the amount of newly generated derivation items. Let $e \in E$ with $\mu(e) = (b_1 b_2, a)$ be some fixed hyperedge.[3] Then, the best derivation with e at its root certainly consists of the best derivations of b_1 and b_2 (up to ties), enforced by the monotony of the underlying monoid **Pr**. Employing the same reasoning, there are two candidates for the second-best derivation with root e: $e(d_1^1, d_2^2)$ and $e(d_1^2, d_2^1)$, with d_1^1 and d_1^2 the best derivations of b_1 resp. b_2, as well as d_1^2 and d_2^2 the corresponding second-best derivations. After all, for $e(d_1^1, d_2^3)$, with d_2^3 the third-best derivation of b_2, we would have

$$\begin{aligned} P(e(d_1^1, d_2^3) \mid G) &= P(d_1^1 \mid G) \cdot P(d_2^3 \mid G) \cdot p(e) \\ &\leq P(d_1^1 \mid G) \cdot P(d_2^2 \mid G) \cdot p(e) && \text{(by monotony of multiplication)} \\ &= P(e(d_1^1, d_2^2) \mid G), \end{aligned}$$

giving us no sufficient reason to prefer $e(d_1^1, d_2^3)$ as candidate to $e(d_1^1, d_2^2)$.

[2] Since, for reasons of efficiency, its implementation will most probably be a *priority queue* (Okasaki, 1999) we also refer to it as the *queue,* but for now it will suffice to think of it as an ordinary set.

[3] Of course, the described idea also works on hyperedges with higher or lower arity, however, for the sake of this exposition, let us first assume two tail nodes.



(a) $i = 5$ (b) $i = 6$

Figure 3.1: Candidates for the i-th best derivation

We can continue this argument as far as necessary, as illustrated by Fig. 3.1, where d_i^j stands for the j-th best derivation of b_i. The large numbers in the different cells denote the *rank* r of the derivation in the cell, where a derivation d is said to have rank r if it is the r-th best derivation of a in the underlying graph.

In Fig. 3.1a, the four best derivations of a have already been determined.[4] This suggests the derivations in the darker shaded cells as candidates for the fifth-best derivation. Let us say this fifth-best one is $e(d_1^2, d_2^2)$. Then we would have to update our set of candidates, now for the sixth-best derivation, by its *neighbors* in two dimensions, $e(d_1^3, d_2^2)$ and $e(d_1^2, d_2^3)$, as shown in Fig. 3.1b.[5]

This observation actually provides us with an effective idea to cope with our problem—we equip every derivation item with the corresponding edge, its rank and *backpointers*, i.e., with the list of the ranks of the derivations it was created from. Such items will be fittingly called *ranked derivation items with backpointers,* or, for brevity's sake, just *ranked derivation items.* Having done so, we can start out the computation with adding the candidate sets for best derivations to the agenda, for each hyperedge coming into question. Their backpointers will necessarily be of the form $1 \cdots 1$ (since best derivations must have been created from best derivations), while their ranks will be undefined at this point, as it will be impossible to determine those until the items have been processed. But as soon as a ranked derivation item is inserted into the chart, we can easily fill in its rank and supply the agenda with its neighbors, as given in the examples above, again for each hyperedge coming into question.

To make it more lucid, let us try to present this idea in the form of the search hypergraph introduced in Def. 3.2.[6] We introduce ranked derivation items with backpointers

$$\left\{ \mathrm{K}(e, r, b) \mid e \in E, r \in \mathbb{N} \cup \{\bot\}, b \in \mathbb{N}^{\mathrm{ar}(e)} \right\}$$

[4] In this example, all of those four have the hyperedge e at their root. This need not be the general case, but it does not invalidate the argument: We can keep such candidate sets for every ingoing hyperedge of a node.

[5] Since $e(d_1^3, d_2^2)$ was already a candidate, effectively we update only with $e(d_1^2, d_2^3)$.

[6] We will, however, omit a complete rigorous definition, due to the fact that the "filling in" of ranks into derivation items after they have been processed does not conveniently fit into the search graph model.

into the node set N' instead of "ordinary" derivation items. The value $\perp$ here denotes yet undetermined ranks of items. The ranked derivation hyperedges are then given by

$$\begin{aligned}&\{\mathrm{build}_0(e) \mid e \in E\}\\ \cup &\left\{\mathrm{build}(e,r,b,i) \mid e \in E, \mathrm{ar}(e) \geq 1, r \in \mathbb{N}, b \in \mathbb{N}^{\mathrm{ar}(e)}, i \in [\mathrm{ar}(e)]\right\},\end{aligned}$$

again replacing the derivation hyperedges in E'. Their connectivity is given by

$$\begin{aligned}\mu'(\mathrm{build}_0(e)) = \big(&\mathrm{O}(\mathrm{hd}(e))\\ &f_{a_1}(1)\cdots f_{a_n}(1),\\ &\mathrm{K}(e,\perp,1\cdots 1)\big)\\ \mu'(\mathrm{build}(e,r,b,i)) = \big(&\mathrm{O}(\mathrm{hd}(e))\ \mathrm{K}(e,r,b)\\ &f_{a_1}(b_1)\cdots f_{a_{i-1}}(b_{i-1})\\ &f_{a_i}(b_i+1)\\ &f_{a_{i+1}}(b_{i+1})\cdots f_{a_n}(b_n),\\ &\mathrm{K}(e,\perp,b_1\cdots b_{i-1}(b_i+1)b_{i+1}\cdots b_n)\big),\end{aligned}$$

for every such hyperedge, with $\mathrm{tl}(e) = a_1 \cdots a_n$, backpointers $b = b_1 \cdots b_n$ and $f_a(k)$ denoting the k-ranked derivation item for head a, for every $k \in \mathbb{N}$ and $a \in N$.

By reading these hyperedge definitions as statements about dependency, as suggested above, they actually speak for themselves: "Presuming we have computed the outside score for a node and all the best-ranked derivation items of the tails of an ingoing hyperedge e, we can devise a rank one candidate for that node."

And: "In order to construct the neighboring candidate for rank $r+1$ in the i-th dimension along hyperedge e, we require the outside score, the according ranked derivation item of rank r, and the ranked derivation item for e's i-th tail which comes in rank right after the i-th backpointer. Moreover, we depend on the other ranked derivation items pointed to."

As for ordinary derivation items, the outside scores serve as heuristic values, going only into the assignments' priorities. Hence the weight and priority functions arise as

$$\begin{aligned}w'_{\mathrm{build}_0(e)}(x_1,\ldots,x_n) &= x_2 \cdots x_n \cdot p(e)\\ w'_{\mathrm{build}(e,r,b,i)}(x_1,\ldots,x_n) &= x_3 \cdots x_n \cdot p(e)\end{aligned}$$

and

$$\begin{aligned}\psi_{\mathrm{build}_0(e)}(x) &= x_1 \cdots x_n \cdot p(e)\\ \psi_{\mathrm{build}(e,r,b,i)}(x_1,\ldots,x_n) &= x_1 \cdot x_3 \cdots x_n \cdot p(e).\end{aligned}$$

Observe how for build-hyperedges the weight of the r-ranked derivation item is discarded in the definitions above. Nevertheless, it has an effect, as its inclusion in the hyperedge's tail nodes prevents the candidate being constructed ahead of its time. Instead, its construction is only triggered when one of its directly neighboring predecessors has been processed.

With the definition of the KA* search hypergraph modified in this way, Alg. 3.1 searches it until no more assignments are on the agenda or k ranked derivation items with the goal

node g as their hyperedge components' heads have been found, while equipping every ranked assignment inserted into the chart with its actual rank. After this, the actual derivations of g have to be reconstructed, this is easily achieved by recursively following the ranked assignments' backpointers and composing the stored hyperedges in the obvious way.

We should note that this lazy reformulation does not change the results of the algorithm: in this way, lazy and non-lazy KA* are equivalent. We will try to ascertain this equivalence. Since they are the only types of items that differ, let us have a look at the relation between derivation items in non-lazy, and ranked derivation items in lazy KA*. As stated before, the creation of the latter is only triggered by the processing of their directly predecessing neighbors. But, eventually, if the algorithm runs that long, they will be processed and the ranked derivation item in question will be inserted into the agenda, along with its computed weight and priority.

So what about the values of these? If we assume both prioritized assignments, once in non-lazy, once in lazy KA*, were created from corresponding assignments with equal weights, it is easy to see that the computed weights and priorities of both coincide. That is, let us assume that $(\mathrm{D}(e(d_1,\ldots,d_k)) = x, q)$ was created from the assignments $(\mathrm{O}(\mathrm{hd}(e)) = x_1)$, as well as $(\mathrm{D}(d_{i+1}) = x_{i+1})$, for $i \in [k]$. Moreover, we presume that $(\mathrm{K}(e, r, b) = x', q')$ was created from assignments $(\mathrm{O}(\mathrm{hd}(e)) = x_1)$, $(\mathrm{K}(e, r', b') = y)$, and $(\mathrm{K}(e_i, r_i, b_i) = x_i)$ for $i \in [k]$.

But then, according to the definitions for the weight function w' in the search hypergraph, we have

$$\begin{aligned} x &= w'_{\mathrm{cat}(e,d_1,\ldots,d_k)}(x_1,\ldots,x_{k+1}) \\ &= x_2 \cdots x_{k+1} \cdot p(e) \\ &= w'_{\mathrm{build}(e,r,b,i)}(x_1, y, x_2, \ldots, x_{k+1}) \\ &= x', \end{aligned}$$

as well as

$$\begin{aligned} q &= \psi_{\mathrm{cat}(e,d_1,\ldots,d_k)}(x_1,\ldots,x_{k+1}) \\ &= x_1 \cdots x_{k+1} \cdot p(e) \\ &= \psi_{\mathrm{build}(e,r,b,i)}(x_1, y, x_2, \ldots, x_{k+1}) \\ &= q', \end{aligned}$$

i.e., their probabilities and priorities agree. An analogous computation can be made for hyperedges of the form build_0. Of course, this short argument does not constitute a proper proof for the equivalence of the two algorithms. However, it should give us some assurance that lazy KA* does not compute something completely different from its non-lazy alternative.

3.3 Implementation of Lazy KA*

With the gist of the idea behind laziness established, we can now begin thinking about its implementation, which is to be achieved in the lazy functional programming language *Haskell* (Peyton-Jones, 2003) and to be integrated into the in-house-developed project *Vanda*, a software platform for research and teaching in the field of Machine Translation.

In a technical report, Shieber, Schabes, and Pereira (1995) introduced the idea of *Deductive Parsing*, a formalism for specifying different parsing strategies (like, e.g., Cocke-Younger-Kasami

or Earley-style parsing) as a set of axioms and deduction rules, formally and notationally similar to logical calculi like natural reasoning. Moreover, they implemented a *meta-interpreter,* facilitating inference from such rules and employing a chart and an agenda as given above, which they brought about in the logic programming language *Prolog*.[7]

It is exactly this formalism of deductive parsing, refined with weights, in which the KA* algorithm is presented in its introductory paper by Pauls and Klein (2009a). Hence, we can draw many ideas from Shieber's et al. Prolog implementation and apply it to the task at hand with Haskell. However, we will also be faced with a few problems specific to our choice of language, after all it immediately suggests itself that a logic programming language like Prolog is more applicable to a formalism similar to natural reasoning than its functional and imperative competitors.

Many ideas presented by Shieber et al. relate to the *trigger item* and associated matters of efficiency. The trigger items (or for the weighted case we deal with, rather the trigger *assignments*) are the items removed from the agenda in each iteration of the algorithm's main loop. They have this specific name because each trigger item possibly triggers the application of an inference rule to it and other "fitting" chart items, to generate a number of new items on the agenda. Since this may happen in every iteration, it is a matter of importance that these other, in some manner compatible items, can be retrieved efficiently from the chart. In the case of Prolog, Shieber et al. could achieve this with predicates mapping items to indices in the chart, however, for our case, where nearly every rule is constructed from one of the graph's hyperedges, we suffice with structures mapping nodes to the hyperedges where they appear in, as well as an organisation of the chart which allows fast access to the different types of assignments. Both can be achieved with Haskell's `Data.Map` datatype, as will be seen later, promising logarithmic lookup times,

However, we will have to deal with matters of symmetry in the implementation of deduction rules. For example, ranked derivation assignment can trigger rules in two ways, as can be seen from the description in Section 3.2: In the first case, the trigger item could stand in the second position of the tail of a build-hyperedge, which means that other fitting ranked items would have to be looked up in order to generate the trigger's neighbors. But the other case might also apply, if the trigger appears in the third position or later, namely that the processing of the trigger assignment allows the construction of new neighbors of *another* ranked derivation item, which was already in the chart. While Prolog's inference system allows Shieber's et al. meta-interpreter to steer clear of such difficulties with elegance, our implementation just splits these rules up into two sub-rules, one for each possible role of the trigger.

Of course this is not the only difference from the Prolog meta-interpreter. Because of Haskell's *purity,* the property that all code is free of side-effects, there arises the problem of how to deal with the state inherent to the algorithm. Of course, one might pass the state—comprised mainly of chart and agenda—around as function arguments, but we chose to hide the current state within a `State` monad.[8] Monads, specifically the list monad `[]`, can be applied to deal with the non-determinism of rule application, too, which is also hidden by Prolog's resolution

[7] For an introduction to Prolog, refer to Hölldobler (2001), for Prolog's history to Colmerauer and Roussel (1993).

[8] Dealing with program and environment state was actually one of the first applications of the concept of monads in Haskell. They are special functors, derived from the field of category theory, and can be used to denote sequential computations with "programmable semicolons," i.e., with modifiable behavior of composition. For a category-theoretic definition, refer to the seminal work by MacLane (1971, ch. VI), for practical applications in Haskell to O'Sullivan, Goerzen, and Stewart (2008, ch. 14).

mechanism. We will see later how the backtracking behaviour implemented by the list monad allows us to give a succinct way to apply rules to all their possible antecedents.

With its disadvantages covered, Haskell's purity has the big advantage that it allows implicit sharing of algebraic datatypes' substructure. This outweighs many of the addressed hassles. Our implementation is split into three modules, `Data`, `State`, and `KAStar`, dealing, respectively, with basic data structures, monadic algorithm state, and the actual rules and execution of the algorithm, each builing up on the previous. Hence, we will divide our exposition into three parts, too.

3.3.1 Data Structures

Items and assignments are of course basic to the whole KA* algorithm.

```
data Assignment v l w i = Inside  (I v l w i) w
                        | Outside (O v l w i) w
                        | Ranked  (K v l w i) w
                          deriving (Show, Eq)
newtype I v l w i = I { iNode :: v } deriving (Show, Eq)
newtype O v l w i = O { oNode :: v } deriving (Show, Eq)
data    K v l w i = K { kNode         :: v
                      , kEdge         :: Hyperedge v l w i
                      , kRank         :: Int
                      , kBackpointers :: [Int]
                      } deriving (Show)
```

Here, the data types `I`, `O` and `K` stand for inside, outside and ranked items. The type `Hyperedge` is imported from the *Vanda* project and introduces the type parametricity over `v`, `l`, `w`, `i`—hypergraph nodes, labels, weights and IDs. For inside and outside items, it suffices to annotate them with their respective node, hence we can use the more efficient *newtype* declaration. Ranked items are constructed from hyperedges, the edge's head nodes, their rank and their backpointers. The `Assignment` algebraic data type equips every item with its priority, also of parametric type `w`.

Next, we define the representation of the algorithm's chart.

```
data Chart v l w i = C { cEdgeMap :: Map v (EdgeMapEntry v l w i)
                      , cBPMap   :: Map (Hyperedge v l w i, Int, Int)
                                        [Assignment v l w i]
                      }
data EdgeMapEntry v l w i = EM { emInside  :: [Assignment v l w i]
                               , emOutside :: [Assignment v l w i]
                               , emRanked  :: Seq (Assignment v l w i)
                               } deriving Show
```

Our chart consists of two maps: the first one, `cEdgeMap`, maps every node in the hypergraph to its associated inside, outside, and ranked items. Since each node has at maximum one inside and outside item, it might have been sensible to use the `Maybe` type constructor instead of a list for them, but since the interface to the chart will export them as lists anyway, and since for this case, the differences are merely cosmetical, we stuck with lists. Ranked assignments were initially

stored in a list, too, but the linear lookup time proved to be too much of a slow-down and we switched the internal representation to the `Data.Sequence` module, promising logarithmic access times.

The reason of existence of the second map, `cBPMap`, is related to optimization, too: later, in the application of the build-rule, we will require efficient lookup of all those ranked derivation assignments containing a hyperedge e whose i-th backpointer has the value j. Of course, the maintenance of this map during the algorithm's execution leads to a certain overhead, but it proved to be of advantage, nevertheless.

Finally, we define the chart's interface with various accessor functions. Since their actual implementation is lengthy, but certainly not very interesting, we stick with their type signatures.

```
insideAssignments :: Ord v ⇒ Chart v l w i → v → [Assignment v l w i]
outsideAssignments :: Ord v ⇒ Chart v l w i → v → [Assignment v l w i]
rankedAssignments :: Ord v ⇒ Chart v l w i → v → [Assignment v l w i]
nthRankedAssignment :: Ord v
                    ⇒ Chart v l w i → v → Int → [Assignment v l w i]
rankedWithBackpointer :: (Ord v, Ord l, Ord w, Ord i)
                      ⇒ Chart v l w i → (Hyperedge v l w i)
                      → Int → Int → [Assignment v l w i]
```

The functions `insideAssignments`, `outsideAssignments` and `rankedAssignments` return the respective lists of assignments for a node in the supplied chart. `nthRankedAssignment` gives back a singleton list containing the ranked assignment to a node with supplied rank, or the empty list if there is no such assignment. Ultimately, `rankedWithBackpointer c e i j` returns those ranked assignments from the chart `c` which contain the hyperedge `e` and whose `i`-th backpointer is equal to `j`. The agenda's definition is quite short by comparison.

```
type Agenda p v l w i = Heap p (w, Assignment v l w i)
```

The module `Data.Heap` serves as an efficient implementation of priority queues. The type parameter `p` can be instantiated with `FstMaxPolicy` or `FstMinPolicy`, toggling whether the order on the weights or its dual order should be used in the comparisons employed by the priority queue. This leads to k-best and k-worst, respectively.

3.3.2 State

Next, we will concern ourselves with the algorithm's state. Central to this lies the `KAStar` monad stack, which comprises not only state, but also configurational parameters, like the supplied hypergraph, etc. Algorithmic state is stored in the `State` monad, while the configuration is read out from the `Reader` monad, which is plugged onto the `State` monad in the form of a *monad transformer*, `ReaderT`. Monad transformers are thoroughly described in O'Sullivan et al. (2008).

```
newtype KAStar p v l w i a = KAStar {
    runK :: ReaderT (KAConfig v l w i) (State (KAState p v l w i)) a
  } deriving ( Monad
             , MonadReader (KAConfig v l w i)
             , MonadState (KAState p v l w i)
             )
```

In order of appearance, the configurational parameters denote the number k of derivations searched for, the underlying weighted hypergraph, its goal node, a consistent heuristic, and two auxiliary data structures, whose function, as mentioned above, is mapping hypergraph nodes to the hyperedges they appear in, together with an integer for the position where they do so.

```
data KAConfig v l w i = KAConfig {
    cfgNumDeriv   :: Int
  , cfgGraph      :: Hypergraph v l w i
  , cfgGoal       :: v
  , cfgHeuristic  :: v → w
  , cfgInEdges    :: M.Map v [(Hyperedge v l w i, Int)]
  , cfgOtherEdges :: M.Map v [(Hyperedge v l w i, Int)]
  }
```

As mentioned above, the algorithm's state is composed from the chart and the agenda. Additionally, we have counters for generation of assignments, as well as for insertions into the chart. These may serve as a measure as to how many redundant assignments the algorithm produces.

```
data KAState p v l w i = KAState {
    stChart          :: Chart v l w i
  , stAgenda         :: Agenda p v l w i
  , stItemsInserted  :: Int
  , stItemsGenerated :: Int
  }
```

We also provide a function to run the monad stack defined above.

```
runKAStar
  :: KAStar p v l w i a
  → Agenda p v l w i → Int
  → Hypergraph v l w i → v → (v → w)
  → M.Map v [(Hyperedge v l w i, Int)]
  → M.Map v [(Hyperedge v l w i, Int)]
  → (a, KAState p v l w i)
runKAStar kst agenda k graph goal heuristic ins others =
    let cfg   = KAConfig k graph goal heuristic ins others
        state = KAState (C M.empty M.empty) agenda 0 0
    in runState (runReaderT (runK kst) cfg) state
```

As shown in the code, the algorithm starts out with an empty chart and agenda, as well as configuration data set by the environment, which is passed by function arguments.

Many of the accessor functions for the data structures from above are reexported in this module, with their computations lifted into the `KAStar` monad. Since the code for these is hardly surprising, we will omit them here and explain their behaviour when they are used, if it does not emerge from their name and context, anyway. One of the more substantial parts of this module is the function `chartInsert`.

```
chartInsert
  :: (Ord v, Ord l, Ord w, Ord i)
  ⇒ Assignment v l w i → KAStar p v l w i (Maybe (Assignment v l w i))
```

```
chartInsert assgmt = do
  enough ← case assgmt of
    (Ranked (K v _ _ _) _)
       → liftM2 (>) ((flip numRanked v) `liftM` chart) numDeriv
    _  → return False
  contained ← chartContains assgmt
  if enough || contained
    then return Nothing
    else do
      incItemsInserted
      bpInsert assgmt
      Just `liftM` eInsert assgmt
  where
    bpInsert a@(Ranked (K _ e _ bps) _) = do
      c ← chart
      let bc' = foldl
                (λ m k → M.insertWith' (++) k [rk c a] m)
                (cBPMap c)
                [(e, bp, val) | bp ← [1 .. length bps]
                              , let val = bps !! (bp - 1)]
      putChart c{cBPMap = bc'}
    bpInsert _ = return ()
    eInsert a = do
      c ← chart
      putChart c{cEdgeMap = M.alter (Just ∘ update c a)
                                    (node a)
                                    (cEdgeMap c)}
      return $ rk c a
    update c a@(Inside _ _)  Nothing   = EM [a] [] empty
    update c a@(Outside _ _) Nothing   = EM [] [a] empty
    update c a@(Ranked _ _)  Nothing   = EM [] []  (singleton $ rk c a)
    update c a@(Inside _ _)  (Just ce) = ce{emInside = [a]}
    update c a@(Outside _ _) (Just ce) = ce{emOutside = [a]}
    update c a@(Ranked _ _)  (Just ce) = ce{emRanked = rk c a
                                                     ◁ emRanked ce}
    rk c (Ranked (K v e r bps) w) =
      Ranked (K v e (succ $ numRanked c v) bps) w
    rk _ x = x
```

For the passed assignment, it checks whether it is already contained in the chart, or, if it is a ranked assignment, if k derivations of the same node were already computed. In this case, we may safely discard it. Else, it is inserted into the chart, i.e. the maps for node-access and backpointer-access are updated accordingly via the subfunctions `eInsert` and `bpInsert`. Alongside, the counter for inserted items is incremented via the monadic function `incItems-Inserted` and the rank of inserted ranked derivation item is determined via the subfunction `rank`. Finally, the item (annotated with its computed rank if applicable) is returned.

Here `empty`, `singleton` and ◁ stand for creation as well as concatenation operations of `Data.Sequence`. Insertion of assignments into the agenda is much simpler:

```
agendaInsert
  :: (Ord v, Ord w, H.HeapItem p (w, Assignment v l w i))
  ⇒ [(w, Assignment v l w i)] → KAStar p v l w i ()
agendaInsert assgnmts = do
  incItemsGenerated $ length assgnmts
  putAgenda =≪ flip (foldl' (flip insert)) assgnmts `liftM` agenda
```

Supplied with a list of generated prioritized assignments, we insert these into the agenda and increase the counter for generated assignments accordingly. Lastly, we provide a means to remove items from the agenda.

```
popAgenda
  :: H.HeapItem p (w, Assignment v l w i)
  ⇒ KAStar p v l w i (Assignment v l w i)
popAgenda = do
  ((p, popped), agenda') ← (fromJust ∘ H.view) `liftM` agenda
  putAgenda agenda'
  return popped
```

Presuming the agenda is non-empty, `popAgenda` removes an assignment with maximal priority, and saves the thus modified agenda. The removed assignment is returned.

3.3.3 Putting it all together

With the groundwork laid in the two previous modules, the implementation of Lazy KA* poses no further trouble.

```
process
  :: (Ord v, Ord w, Ord i, Ord l, H.HeapItem p (w, Assignment v l w i))
  ⇒ KAStar p v l w i (Maybe (Assignment v l w i))
process = do
  d ← done
  if d then return Nothing
       else do -- agenda != empty
         popped ← popAgenda
         trigger ← chartInsert popped
         case trigger of
           Nothing → process
           _       → return trigger
  where done = do
          e ← H.isEmpty `liftM` agenda
          l ← liftM2 numRanked chart goal
          k ← numDeriv
          return $ e || l ≥ k
```

The function `process` removes assignments from the agenda and tries to insert them into the chart (via `chartInsert`), until it finds an assignment that was not already inserted and might

therefore act as a trigger, which is then returned. If the agenda is empty, or k derivations of the goal node have been found before that, the function returns `Nothing`.

The function `process` is employed in `kastar`, which models the whole of KA*.

```
kastar
  :: (Num w, Ord v, Ord w, Ord l, Ord i,
      H.HeapItem p (w, Assignment v l w i))
  ⇒ Agenda p v l w i → Hypergraph v l w i → v → (v → w) → Int
  → [(T.Tree (Hyperedge v l w i), w)]
kastar agenda graph g h k
  = reverse ∘ mapMaybe (traceBackpointers res) $ rankedAssignments res g
  where (res, info)   = runKAStar kst agenda k graph g h ins others
        (ins, others) = edgesForward graph
        kst = initialAssignments ≫= agendaInsert ≫ loop
        loop = do
          m ← process
          case m of
            Nothing        → chart
            (Just trigger) → newAssignments trigger ≫= agendaInsert
                           ≫ loop
```

The function starts out with generating the auxiliary data structures `cfgInEdges` and `cfgOtherEdges` by invocation of `edgesForward`. Afterwards, the initial assignments are inserted into the chart in `kst`, and then `loop` alternately pops new trigger assignments via `process` and inserts their consequences into the agenda. When the work is done, `process` returns `Nothing`, the derivations of the goal node are reconstructed with `traceBackpointers`, brought into ascending order with `reverse` and emitted.

Since the function is still abstract in the ordering properties of the agenda, we will have to introduce two specializations: one, `kbest`, where the agenda prioritizes higher weights, and the other, `kworst`, where the contrary applies.

```
kbest, kworst
  :: (Num w, Ord v, Ord w, Ord l, Ord i)
  ⇒ Hypergraph v l w i → v → (v → w) → Int
  → [(T.Tree (Hyperedge v l w i), w)]
kbest  = kastar (H.empty :: H.MaxPrioHeap w (Assignment v l w i))
kworst = kastar (H.empty :: H.MinPrioHeap w (Assignment v l w i))
```

Let us view how the initial assignments, with which the agenda is populated at the beginning of the algorithm, are created:

```
initialAssignments
  :: (Num w, Ord v, Eq l, Eq i)
  ⇒ KAStar p v l w i [(w, Assignment v l w i)]
initialAssignments = do
  g ← graph
  h ← heuristic
  return $ map (ins h) ∘ filter ((== 0) ∘ length ∘ eTail)
      ∘ concat ∘ M.elems ∘ edgesM $ g
```

```
    where ins h e = let w = eWeight e
                        p = w * (h ∘ eHead $ e)
                    in (p, Inside (I ∘ eHead $ e) w)
```

As detailed in Alg. 3.1, at first we only process hyperedges of arity zero. Their weights constitute the corresponding assignments' weights, whose priorities are computed with the external heuristic `h`.

All that is left to be done is to describe the rules used to generate new assignments from the trigger. Every rule is implemented as a function, with the triggering assignment among its arguments, as well as various auxiliary data structures, used for optimization. They all return a list of new prioritized assignments, hence we can employ list-monadic computations in their bodies, generating all possible instantiations of this rule at once, as well as dealing with failure, behind the scenes.

The rules are called, depending on the type of the triggering assignment, by the function `newAssignments`. Let us start out with the switch rule.

```
switchRule :: Chart v l w i
           → v
           → Assignment v l w i
           → [(w, Assignment v l w i)]
switchRule c g trigger = do
  guard $ isInside trigger && node trigger == g
  return $! (weight trigger, Outside (O g) 1)
```

If the trigger was an inside item containing the goal node, we can switch to the computation of outside items, i.e., we emit the outside item, with weight 1.0 and priority determined from the weight of the triggering assignment. Next, we will have a look at the inside rule.

```
inRule :: Chart v l w i → (v → w) → Assignment v l w i
       → (Hyperedge v l w i, Int) → [(w, Assignment v l w i)]
inRule c h trigger (e, r) = do
  ibsl ← mapM (insideAssignments c) ∘ take r $ eTail e
  ibsr ← mapM (insideAssignments c) ∘ drop (r + 1) $ eTail e
  let ibs = ibsl ++ [trigger] ++ ibsr
      w   = eWeight e * (product ∘ map weight $ ibs)
      p   = h (eHead e) * w
  return $! (p, Inside (I (eHead e)) w)
```

The rule is supplied with the trigger item `trigger`, together with an entry `(e,r)` from `cfgInEdges`, detailing a hyperedge `e` whose tail the trigger's associated node appears in, together with an integer `r` specifying the position. Then the chart is queried for compatible inside assignments lying "around" this position. If that is so, we can generate a new prioritized assignment, with priority taken from the external heuristic `h`. This rule can only be triggered by inside items. The out-rule is slightly more complex:

```
outRule :: Chart v l w i → Assignment v l w i
        → (Hyperedge v l w i, Int)
        → [(w, Assignment v l w i)]
outRule c trigger (e, r) = do
```

```
  (oa, ibs)
    ← if r == 0
       then do
         guard $ isOutside trigger
         liftM2 (,) [trigger] (mapM (insideAssignments c) (eTail e))
       else do
         guard $ isInside trigger
         ibsl ← mapM (insideAssignments c) ∘ take (r - 1) $ eTail e
         ibsr ← mapM (insideAssignments c) ∘ drop r $ eTail e
         liftM2 (,) (outsideAssignments c $ eHead e)
                    [ibsl ++ [trigger] ++ ibsr]
  i ← [0 .. (length ibs - 1)]
  let w = eWeight e * weight oa
          * (product ∘ map weight $ take i ibs)
          * (product ∘ map weight $ drop (i + 1) ibs)
      p = w * weight (ibs !! i)
  return $! (p, Outside (0 (eTail e !! i)) w)
```

Again, the rule is provided with the triggering assignment, a hyperedge, and the position of the assignment's node in this edge. Since the processing of inside and outside assignments is interleaved, as described above, the rule can be triggered by both types of assignments. If the trigger is an outside assignment of the edge's head node (`r = 0`), compatible inside assignments are queried from the chart, else if it is an inside assignment of one of the tail nodes (`r > 0`), we look for fitting inside assignments around the trigger, as with the inside rule, and an outside assignment for the edge's head. The rule then goes on and constructs an outside item for each of the hyperedge's tail nodes, via monadic selection of `i`.

Next, we consider `buildRule0`:

```
buildRule0 :: Chart v l w i → Assignment v l w i
           → (Hyperedge v l w i, Int) → [(w, Assignment v l w i)]
buildRule0 c trigger@(Outside _ _) (e, 0) = do
  as ← mapM (flip (nthRankedAssignment c) 1) (eTail e)
  let w   = eWeight e * (product ∘ map weight $ as)
      p   = w * weight trigger
      bps = map rank as
  return $! (p, Ranked (K (eHead e) e 0 bps) w)
buildRule0 c trigger@(Ranked _ _) (e, r) = do
  guard $ r ≠ 0 && rank trigger == 1
  oa  ← outsideAssignments c $ eHead e
  asl ← zipWithM (nthRankedAssignment c) (take (r - 1) $ eTail e)
                 (repeat 1)
  asr ← zipWithM (nthRankedAssignment c) (drop r $ eTail e) (repeat 1)
  let assmts  = asl ++ [trigger] ++ asr
      w       = eWeight e * (product ∘ map weight $ assmts)
      p       = w * weight oa
      bps     = map rank assmts
  return $! (p, Ranked (K (eHead e) e 0 bps) w)
```

```
buildRule0 _ _ _ = []
```

This rule's purpose lies in the construction of new ranked derivation assignments, with backpointers only to best-ranked assignments (i.e., every backpointer's value is one). Thus, one can think of the rule as initialization of the structures used for candidate search as described in Section 3.2. Again, because processing of the different types of assignments is not clearly divided into distinct phases, the triggering assignment may be an outside or a ranked assignment. In the first case, we query the chart for best-ranked assignments of the tail nodes of the supplied hyperedge, and construct the corresponding new ranked assignment. In the other case, the triggering item is required to have rank one, and is then combined with compatible best-ranked assignments along the supplied hyperedge, yielding a new ranked assignment to be inserted into the agenda.

Note that when we disregard the above case distinction on the type of the trigger, the rule can be thought of as an implementation of the build_0-rule introduced in Section 3.2, once with an outside assignment as trigger, and once with a best-ranked derivation assignment.

The following two rules, `buildRuleL` and `buildRuleR`, implement together the functionality of the build-rule, again with a case distinction on the role of the triggering assignment.

```
buildRuleL :: Chart v l w i → Assignment v l w i
           → M.Map v [(Hyperedge v l w i, Int)]
           → [(w, Assignment v l w i)]
buildRuleL c trigger@(Ranked _ _) inEdges
  = concatMap rule (M.findWithDefault [] (node trigger) inEdges)
  where
    rule (e, s) = do
      Ranked (K _ e' _ bps) _ ← rankedWithBackpointer c e (s + 1)
                                  (rank trigger - 1)
      asl ← take s `liftM`
               zipWithM (nthRankedAssignment c) (eTail e) bps
      asr ← drop (s + 1) `liftM`
               zipWithM (nthRankedAssignment c) (eTail e) bps
      oa  ← outsideAssignments c (eHead e)
      let assmts  = asl ++ [trigger] ++ asr
          w       = eWeight e * (product ∘ map weight $ assmts)
          p       = w * weight oa
          bps     = map rank assmts
      return $! (p, Ranked (K (eHead e) e 0 bps) w)
buildRuleL _ _ _ = []
```

The above rule is triggered by an r-ranked derivation assignment. It updates all assignments in the chart possessing a backpointer to the corresponding $(r-1)$-ranked assignment with its trigger, thus generating new neighboring candidates to be inserted into the agenda. For reasons of optimization, `buildRuleL` is directly supplied with the `inEdges` data structure. Moreover, observe how the function utilizes `rankedWithBackpointer` for efficient lookup of assignments with backpointers directly below the trigger's rank. The inherent backtracking behaviour of the list monad ensures that the rule is applied to every ranked assignment with fitting backpointers.

The rule `buildRuleR` fulfills diametral purposes.

```
buildRuleR :: Chart v l w i → Assignment v l w i
           → [(w, Assignment v l w i)]
buildRuleR c trigger@(Ranked (K _ e _ bps) _) = do
  r ← [1 .. length bps]
  let bps' = zipWith (+) bps (unit (length bps) r)
  assmts ← zipWithM (nthRankedAssignment c) (eTail e) bps'
  oa ← outsideAssignments c (eHead e)
  let w = eWeight e * (product ∘ map weight $ assmts)
      p = w * weight oa
  return $! (p, Ranked (K (eHead e) e 0 bps') w)
  where
    unit n r = replicate (r - 1) 0 ++ [1] ++ replicate (n - r) 0
buildRuleR _ _ = []
```

Its trigger is a ranked assignment with backpointers. For each of its backpointers, the chart is queried for the next-best ranked derivation assignment. This is achieved by pointwise addition of all possible unit vectors (i.e., lists of the form `[0,...,0,1,0,...,0]`) to the item's backpointers and querying the chart for the appropriate ranked assignments. If such assignments are in the chart, we can construct a neighboring candidate in the unit vector's respective dimension. With these rules, the exposition of Lazy KA*'s implementation is concluded.

4 Grammars and Hypergraphs

Through the following chapter, we will occupy ourselves with the formalism of probabilistic tree insertion grammars, used in the field of Natural Language Processing. We will develop a notion of how to transform such grammars into probabilistic hypergraphs, and how to equip these hypergraphs with latent annotation symbols. This allows use of the State-Split procedure, facilitating accurate learning of probabilistic hypergraphs from supplied treebanks of natural language samples.

In the course of this, we will detail the Expectation-Maximization algorithm for maximum-likelihood estimation of complete data, given some incomplete data, and argue why the Inside-Outside algorithm can be seen as a dynamic programming instance of EM.

4.1 Probabilistic Tree Insertion Grammars

Probabilistic tree insertion grammars (ptig) are seeing increasing use in the fields of Natural Language Processing and Machine Translation (cf. Nesson, Shieber, and Rush, 2006). They can be thought of as special instances of *tree adjoining grammars (tag)*, first introduced by Joshi (1985). Let us consider those, first of all.

In contrast to the well-known context-free grammars, the basic building blocks of tree adjoining grammars are not mere strings over terminal and non-terminal symbols, but *trees*. Accordingly, the products of derivations of tags are trees instead of strings, too. These trees are called the *derived trees*, or also *full-parse trees*, of the tree adjoining grammar in question.

To be more precise, a tag consists of two sets of (unranked) trees over terminals and non-terminals, called the set of *initial trees* and the set of *auxiliary trees*, with the union of these sets simply referred to as the set of the grammar's *trees*. These two types of trees correspond to the two possible derivation operations in the formalism, namely *substitution* as well as *adjoining*.

Substitution facilitates handling the discontinuity introduced in natural languages by such phrases as "drink *sb.* under the table", "drive *sb.* mad", etc.

It takes place on non-terminal leaf nodes of the grammar's trees. During this operation, the non-terminal is replaced by a tree derived from an initial tree, with its root node equal to the replaced leaf node. An example for substitution is displayed in Fig. 4.1. There, we have two trees (possibly already derived from other trees), with English words as terminals and part-of-speech tags (cf. Manning and Schütze, 1999, ch. 3.1) as non-terminal symbols. The upper tree, with the yield "drink *(sb.)* under the table", has a leaf non-terminal NP, which is ready for substitution. By convention, such nodes are marked with a downward arrow (↓) in figures showing the trees in question.

Since the lower tree, with the yield "old Jim", has NP as its root, and was derived from an initial tree (at least we pretend so in this example), it can be substituted into the upper one, deriving the tree on the right, with yield "drink old Jim under the table".

Adjoining, on the other hand, allows to "plug" modifiers into already derived trees. This may be used to add adjectives to noun phrases or adverbs to verb phrases, arriving at more

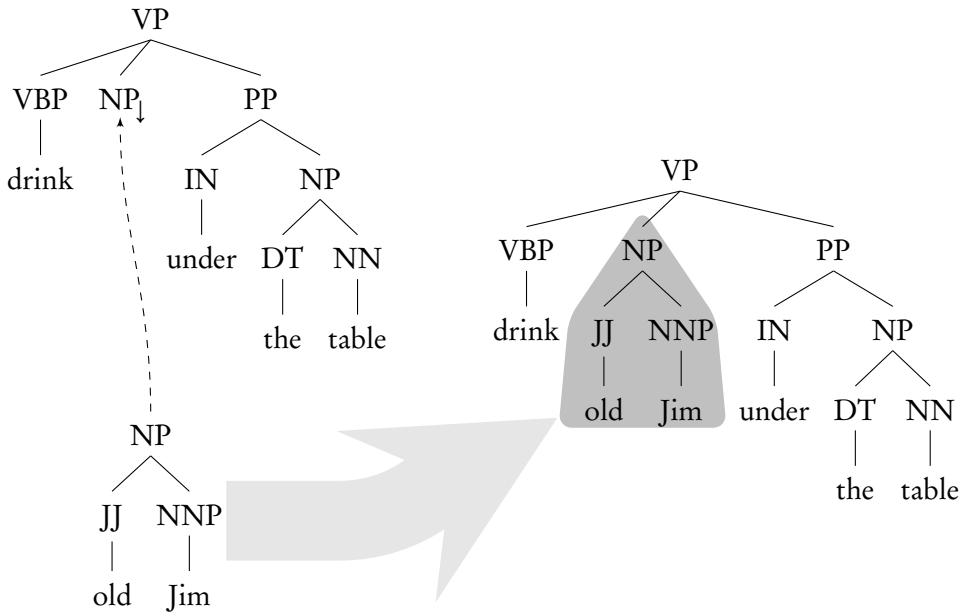


Figure 4.1: Substitution of initial trees

complex noun or verb phrases.

For this plugging in to take effect, every auxiliary tree has a uniquely determined leaf node with the same label as its root, called its *foot node.* Adjoining is then allowed for a non-leaf position w of a tree t, designated as an *adjoining site*, together with an auxiliary tree t' with root label equal to the label at w. It is performed by

(i) replacing the foot node of t' with the subtree of t at position w, arriving at the tree t'', and then

(ii) replacing the subtree of t at position w with t''.

For an illustrative example of adjoining, let us turn our attention to Fig. 4.2. There, we have a tree with yield "the cat sat on the mat" and an auxiliary tree with yield "yesterday." As customary for tags, the foot node of the auxiliary tree is marked with a star ($*$). We adjoin the auxiliary tree into the former tree's only position labeled with PP (prepositional phrase) by the approach from above: The auxiliary tree is substituted for the former's PP node, while its foot node is replaced by the subtree at PP (whose yield is "on the mat"). We arrive at the tree displayed on the right, yielding "the cat sat on the mat yesterday."

Similar to probabilistic context-free grammars, tags can also be extended with probabilities, arriving at *probabilistic tree adjoining grammars (ptag).* There are manifold ways to add probabilities to tags, compare Schabes (1992), but we will content ourselves with providing every tree with the probability of it being used at a certain point in a derivation, as well as every designated adjoining site of a tree with the probability that adjoining will take place on it. Note that thus we can enforce adjoining on a certain site, by setting the corresponding probability to one.

As argued by Nesson et al. (2006), the probability models induced by ptags have a definite advantage over the ones induced by pcfgs: since for ptags, we assign probabilities to whole

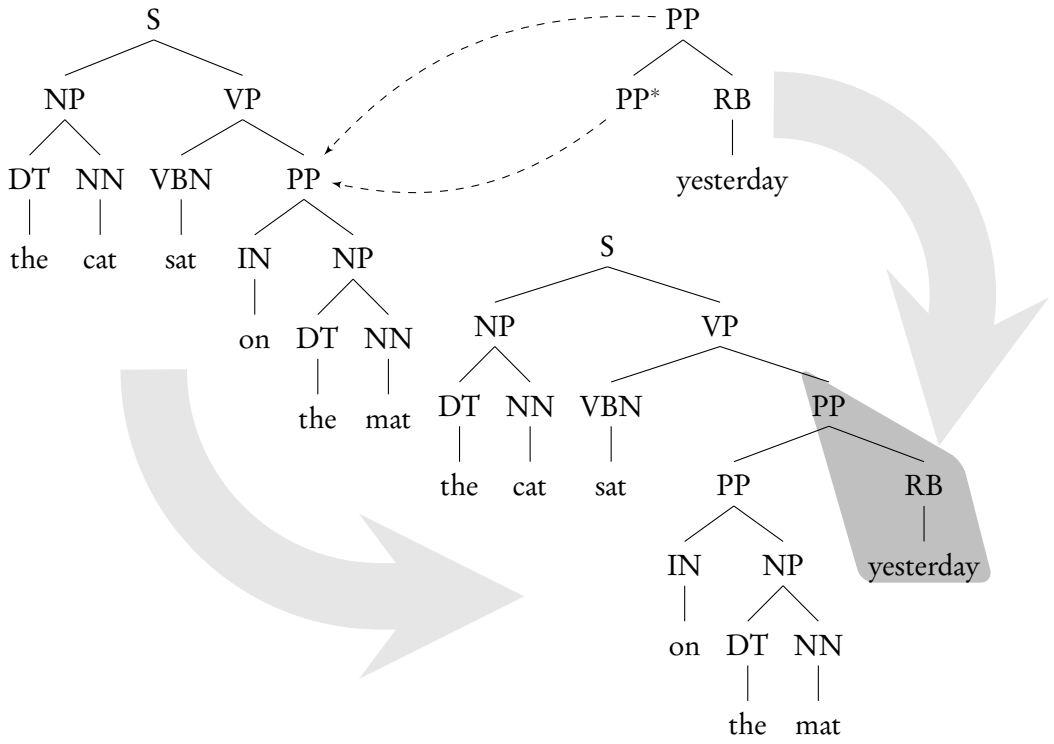


Figure 4.2: Adjoining of an auxiliary into an initial tree

subtrees of the resulting full-parse tree instead of separate rules, there is the possibility to prefer certain idiomatic phrasal structures over the components they are built from—e.g., going back to the example from above, the idiom "drink *sb.* under the table" might have a significantly higher occurrence than the phrase "under the table", depending on the domain of discourse. With ptags, we can easily adapt to this by modifying the tree's probability, but with pcfgs, there is no such possibility.[1]

Finally, since parsing of tags is in $O(n^6)$ (in the length of input strings as well as non-terminal symbols), we will restrict the (probabilistic) tree adjoining grammars used in the following to *(probabilistic) tree insertion grammars ((p)tig)*, for which there are parsing algorithms in $O(n^3)$, as given by Schabes and Waters (1994).

Tree insertion grammars achieve this higher efficiency by disallowing *wrapping trees*, i.e., trees derived from an auxiliary tree whose foot node has other leaf nodes to its right as well as to its left. Hence we will only allow those trees whose foot node is at the left (called *right*-auxiliary trees), and those whose foot node is at the right of their yield (these are called *left*-auxiliary trees). Adherence to this constraint has to be enforced throughout the whole derivation of a full-parse tree. For an example for wrapping trees, refer to Fig. 4.3. There, adjoining of the right-auxiliary tree (on the left) into the left-auxiliary tree (in the middle) results in the wrapping tree on the right.

The occurrence of wrapping trees can be prevented by explicitly allowing resp. disallowing

[1] Compare also to the arguments regarding tree substitution grammars by Manning and Schütze (1999, p. 448), which can be carried over to ptigs.

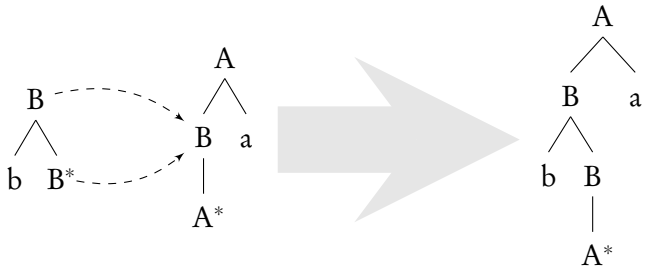


Figure 4.3: Production of a wrapping tree by unwise adjoining

adjoining on every tree position via labeling the corresponding nodes with the symbols L and R. These signal possibility of adjoining of a left, resp. right auxiliary tree. A position on the spine of a left-auxiliary (resp. right-auxiliary) tree must then not be labeled with R (resp. L), or else adjoining might result in a wrapping tree.

We now have the necessary tools to give a definition for probabilistic tree insertion grammars. The subsequent definitions follow the formalization of *probabilistic synchronous tree insertion grammars* by DeNeefe, Knight, and Vogler (2010), differing only in the omission of the synchronous processing. First of all, we will give an idea of what the *rules* of ptigs look like.

Definition 4.1. Presuming two disjoint sets, T and N, of *terminal* and *non-terminal symbols,* a *substitution rule* is a tuple $r = (\zeta, W, P^r_{\text{adj}})$ where

- $\zeta \in U_N(T)$, called the *tree* of this rule,
- $W \subseteq \text{nlv}(\zeta) \times \{L, R\}$, denoting positions and allowed directions for possible adjunctions in the tree above, called the set of *potential adjoining sites*,
- $P^r_{\text{adj}} \colon W \to [0, 1]$, denoting the *adjoining probability* for each site.

Similarly, an *auxiliary rule* is a tuple $r = (\zeta, W, *, P^r_{\text{adj}})$ where ζ, W and P^r_{adj} are as above and we demand additionally:

- The position $* \in \text{lv}_N(\zeta)$, called the *foot node* of this rule, satisfies $\zeta(\varepsilon) = \zeta(*)$ as well as $* \neq \varepsilon$. The sequence of prefixes of $*$, ordered by ascending length, is called the tree's *spine*.
- $*$ is the left- or rightmost leaf, i.e. for every position $w \in \text{lv}(\zeta)$, we have $* \leq_* w$, resp. $w \leq_* *$, according to the lexicographic order $\leq_*$ over $\mathbb{N}^*$, defined in Def. 2.2 above. In the first case, we call the rule *R-auxiliary*, in the second case *L-auxiliary*.
- there must be at least one leaf apart from the foot node: $|\text{lv}(\zeta)| \geq 2$,
- for every potential adjoining site $(w, \rho) \in W$, if w lies on the spine of ζ and r is L-auxiliary, then $\rho = L$. Analogously, if the same condition applies and r is R-auxiliary, then $\rho = R$.

For every substitution rule $r = (\zeta, W, P^r_{\text{adj}})$ and auxiliary rule $r = (\zeta, W, *, P^r_{\text{adj}})$, we define the rule's *root category* as $\text{rc}(r) = \zeta(\varepsilon)$.

Note that these two types of rules are basically nothing else than the initial and auxiliary trees from above, supplied with possible adjoining sites, and the directions for these adjunctions, which prevent the creation of wrapping trees, as outlined above. Moreover, every potential adjoining site receives a probability for *activation*, i.e., the likelihood it will be used for adjoining in the derivation of a full-parse tree later on.

Probabilistic tree insertion grammars comprise then nothing more than such rules, together with their probabilities of application, and of course the sets of terminal and non-terminal symbols.

Definition 4.2. A *probabilistic tree insertion grammar (ptig)* is a tuple $\mathcal{G} = (N, T, S, \mathcal{S}, \mathcal{A}, P)$ where

- N and T are disjoint sets, called, resp., the set of *non-terminals* and *terminals*,
- $S \in N$ is called the *start non-terminal*,
- $\mathcal{S}$ and $\mathcal{A}$ are sets of substitution, resp. auxiliary rules with non-terminals N and terminals T,
- and $P \colon \mathcal{S} \cup \mathcal{A} \to [0, 1]$ assigns probabilities to rules such that for every $A \in N$ and $x \in \{L, R\}$:

$$\sum_{\substack{r \in \mathcal{S} \\ \mathrm{rc}(r) = A}} P(r) = 1 \qquad\qquad \sum_{\substack{r \in \mathcal{A},\, \mathrm{rc}(r) = A \\ r \text{ is } x\text{-auxiliary}}} P(r) = 1,$$

  provided that in each case the number of summands is greater than zero.

Moreover, we demand that derivations of such grammars can not become "stuck": for every tree ζ of a rule $r \in \mathcal{S} \cup \mathcal{A}$, and every non-terminal leaf position $w \in \mathrm{lv}_N(\zeta)$, there must be a substitution rule $s \in \mathcal{S}$ with $\mathrm{rc}(s) = \zeta(w)$. Similarly, for every potential adjunction site (w, ρ) of r, there must be a ρ-auxiliary rule $a \in \mathcal{A}$ with $\mathrm{rc}(a) = \zeta(w)$.

Probabilistic tree insertion grammars with no auxiliary rules ($\mathcal{A} = \emptyset$) are called *probabilistic tree substitution grammars (ptsg)*. Since they are a special case of ptigs, the methods described for ptigs throughout this work carry over naturally to ptsgs.

To give an intuition of what ptigs and their application may look like, let us consider a short example.

Example 4.3. The ptig $\mathcal{G} = (N, T, S, \mathcal{S}, \mathcal{A}, P)$ under consideration allows us to derive parse trees over a comparably small domain of discourse, but it illustrates most phenomena associated with ptigs. It is composed of

- non-terminal symbols

$$N = \{\mathrm{S}, \mathrm{NP}, \mathrm{DT}, \mathrm{NN}, \mathrm{VP}, \mathrm{VBZ}, \mathrm{PP}, \mathrm{IN}, \mathrm{PRP}, \mathrm{JJ}\},$$

  a set of part-of-speech tags in the style of the Penn treebank (Marcus, Santorini, and Marcinkiewicz, 1994),

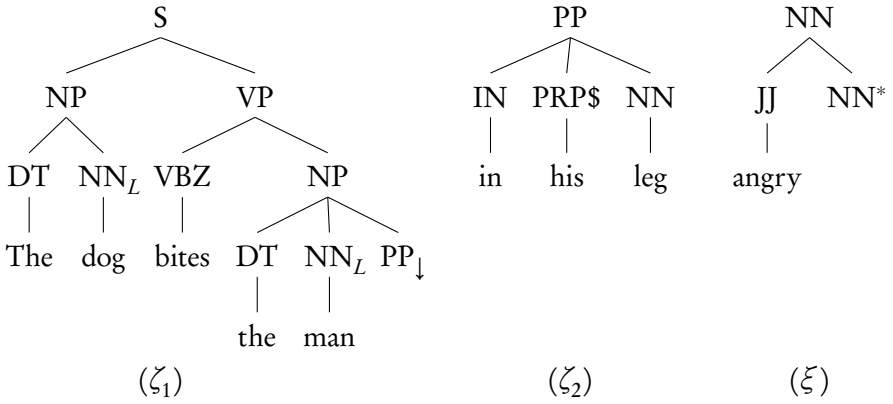


Figure 4.4: Initial and auxiliary trees of an example ptig

- a set of English words as terminal symbols,

  $$T = \{\text{The}, \text{the}, \text{man}, \text{dog}, \text{bites}, \text{in}, \text{his}, \text{leg}, \text{angry}\},$$

- substitution rules given by $\mathscr{S} = \{s_1, s_2\}$ with

  $$s_1 = (\zeta_1, \{(12, L), (222, L)\}, P^{s_1}_{\text{adj}}),$$

  with adjoining probabilities, say, $P^{s_1}_{\text{adj}}(12, L) = 0.5$ as well as $P^{s_1}_{\text{adj}}(222, L) = 0.3$ and

  $$s_2 = (\zeta_2, \emptyset, P^{s_2}_{\text{adj}}),$$

  where, since the rule has no adjoining sites, $P^{s_2}_{\text{adj}} = \emptyset$,

- and one single auxiliary rule $\mathscr{A} = \{a\}$ where

  $$a = (\xi, \emptyset, *, P^{a}_{\text{adj}})$$

  with the position of the foot node $* = 2$ and, as above, $P^{a}_{\text{adj}} = \emptyset$.

- Since our grammar is rather small, there is not much choice for rule probabilities—we have

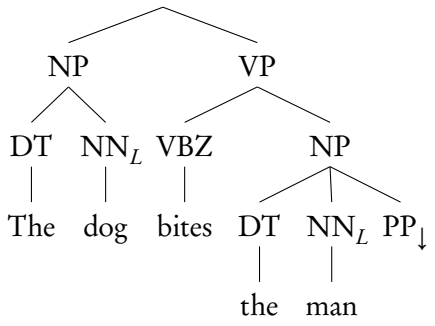
  $$P(s_1) = P(s_2) = P(a) = 1.$$

The trees ζ_1, ζ_2 and ξ are as displayed in Fig. 4.4. With this grammar we can, e.g., derive a tree which yields the sentence *"The angry dog bites the man in his leg."* An example derivation of such a tree is displayed in Fig. 4.5.

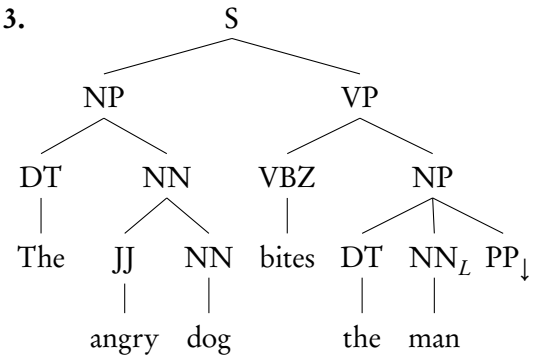
One could now formally define such derivations as in the example above, directly for this grammar formalism, as given by DeNeefe, Knight, and Vogler. However, we will take a detour over probabilistic hypergraphs, defining a transformation from ptigs into phgs. The derivations

1. $S_{\downarrow}$ *(Start symbol S)*

2. *(Substituting S with ζ_1)*



3. *(Adjoining ξ into position 12)*



4. *(Substituting ζ_2 into position 223)*

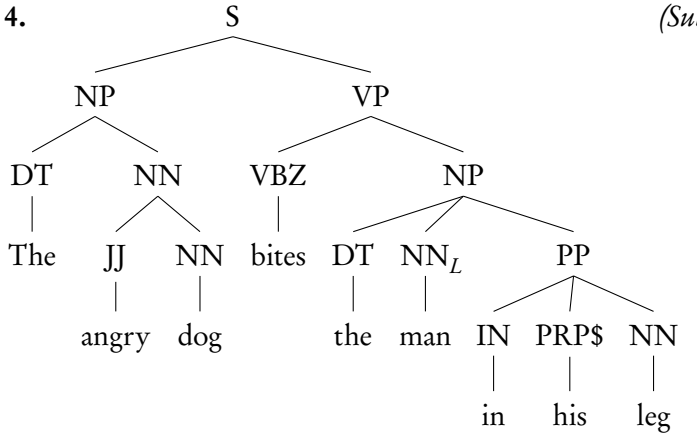


Figure 4.5: Deriving a tree

in the created hypergraph can be thought of as a road-map, describing what substitutions and adjoining operations are applied at which sites in a corresponding derivation in the ptig formalism.[2]

Definition 4.4. Assume a ptig $\mathcal{G} = (N, T, S, \mathcal{S}, \mathcal{A}, P)$. Then we define its generated probabilistic hypergraph as $\mathrm{HG}(\mathcal{G}) = (N', E, \mu, g, p)$ with

- the set of nodes containing the non-terminals of the grammar as well as rules annotated with one of their possible adjoining sites:

$$N' = \{\mathrm{rc}(r) \mid r \in \mathcal{S}\} \cup \{(r, w) \mid r = (\zeta, W, *, P^r_{\mathrm{adj}}) \in \mathcal{A}, w \in W\} \\ \cup \{(r, w) \mid r = (\zeta, W, P^r_{\mathrm{adj}}) \in \mathcal{S}, w \in W\},$$

- edges comprising substitution edges, which denote substitution into other trees and activation of adjoining sites, as well as adjoining edges, which allow adjoining into other trees, but also further substitution and activation of other adjoining sites:

$$E = \{\mathrm{subst}(r, u) \mid r = (\zeta, W, P^r_{\mathrm{adj}}) \in \mathcal{S}, u \in W^*\} \\ \cup \{\mathrm{adj}(r, u, s, v) \mid r = (\zeta, W, *, P^r_{\mathrm{adj}}) \in \mathcal{A}, u \in W^*, s = (\xi, V, P^s_{\mathrm{adj}}) \in \mathcal{S}, \\ v = (w, \rho) \in V, \xi(w) = \zeta(*), r \text{ is } \rho\text{-auxiliary}\} \\ \cup \{\mathrm{adj}(r, u, s, v) \mid r = (\zeta, W, *, P^r_{\mathrm{adj}}) \in \mathcal{A}, u \in W^*, s = (\xi, V, *', P^s_{\mathrm{adj}}) \in \mathcal{A}, \\ v = (w, \rho) \in V, \xi(w) = \zeta(*), r \text{ is } \rho\text{-auxiliary}\},$$

  where for each of the $u = (w_1, \rho_1) \cdots (w_k, \rho_k) \in W^*$, denoting the activated substitution sites, we demand that, for all $i, j \in [k]$,

  (i) the symbols of u are ordered by their first components: $i < j$ implies $w_i \leq_* w_j$,

  (ii) and they are unique in u: $w_i = w_j$ and $\rho_i = \rho_j$ implies $i = j$,

- for edge connectivity, we will have non-terminals as head nodes of substitution edges, while the head nodes of an adjoining edge is the pair of a rule and some adjoining site of this rule which allows adjoining:

$$\mu\big(\mathrm{subst}(r, u_1 \cdots u_k)\big) = \big(\mathrm{yield}_N(\zeta)(r, u_1) \cdots (r, u_k), \zeta(\varepsilon)\big),$$

  where $r = (\zeta, W, P^r_{\mathrm{adj}})$ and

$$\mu\big(\mathrm{adj}(r, u_1 \cdots u_k, s, v)\big) = \big(\mathrm{yield}'_N(\zeta, *)(r, u_1) \cdots (r, u_k), (s, v)\big),$$

  where $r = (\zeta, W, *, P^r_{\mathrm{adj}})$, [3]

- its goal node is the grammar's start non-terminal, $g = S$,

[2] In this, they correspond to the *derivation trees* specified by Joshi and Schabes (1991, p. 7), which should not be confused with *derived trees*, i.e., the trees generated by a ptig.

[3] Note that the conditions for u from the point above induce a canonical order for the tail nodes of a hyperedge: there does not arise any ambiguity in the order of adjoining sites. However, we can distinguish between the order of left- and right-adjoining on some site labeled both with L and R.

- and edge weights are given as

$$p\Big(\operatorname{subst}\big((\zeta, W, P^r_{\mathrm{adj}}), u\big)\Big) = P(r)\cdot\prod_{q\in\mathrm{set}(u)} P^r_{\mathrm{adj}}(q)\cdot\prod_{q\in W\setminus\mathrm{set}(u)}(1-P^r_{\mathrm{adj}}(q))$$

$$p\Big(\operatorname{adj}\big((\zeta, W, *, P^r_{\mathrm{adj}}), u, s, v\big)\Big) = P(r)\cdot\prod_{q\in\mathrm{set}(u)} P^r_{\mathrm{adj}}(q)\cdot\prod_{q\in W\setminus\mathrm{set}(u)}(1-P^r_{\mathrm{adj}}(q))$$

  with $\mathrm{set}(u) = \{u_i \mid i \in [k]\}$ for $u = u_1 \cdots u_k$.

Example 4.5. Let us illustrate this transformation into hypergraphs with the aid of the ptig $\mathscr{G} = (N, T, S, \mathscr{S}, \mathscr{A}, P)$ from Ex. 4.3. We receive a phg $\mathrm{HG}(\mathscr{G}) = (N', E, \mu, g, p)$ with

$$N' = \big\{\mathrm{S}, \mathrm{PP}, \big(s_1, (12, L)\big), \big(s_1, (222, L)\big)\big\},$$

where s_1 and s_2 are the respective substitution rules from the example. The hypergraph's edges comprise

$$\begin{aligned} E = \{&\operatorname{subst}\big(s_1, \varepsilon\big), \operatorname{subst}\big(s_1, (12, L)\big), \operatorname{subst}\big(s_1, (222, L)\big), \operatorname{subst}\big(s_1, (12, L)(222, L)\big),\\ &\operatorname{adj}\big(a, \varepsilon, s_1, (12, L)\big), \operatorname{adj}\big(a, \varepsilon, s_1, (222, L)\big)\} \end{aligned}$$

with head and tail nodes given by

$$\begin{aligned} &\mu\big(\operatorname{subst}\big(s_1, \varepsilon\big)\big) = \big(\mathrm{PP}, \mathrm{S}\big)\\ &\mu\big(\operatorname{subst}\big(s_1, (12, L)\big)\big) = \big(\mathrm{PP}\,(s_1, (12, L)), \mathrm{S}\big)\\ &\mu\big(\operatorname{subst}\big(s_1, (222, L)\big)\big) = \big(\mathrm{PP}\,(s_1, (222, L)), \mathrm{S}\big)\\ &\mu\big(\operatorname{subst}\big(s_1, (12, L)(222, L)\big)\big) = \big(\mathrm{PP}\,(s_1, (12, L))(s_1, (222, L)), \mathrm{S}\big)\\ &\mu\big(\operatorname{adj}(a, \varepsilon, s_1, (12, L))\big) = \big(\varepsilon, (s_1, (12, L))\big)\\ &\mu\big(\operatorname{adj}(a, \varepsilon, s_1, (222, L))\big) = \big(\varepsilon, (s_1, (222, L))\big), \end{aligned}$$

and their probabilities are

$$\begin{aligned} &p\big(\operatorname{subst}\big(s_1, \varepsilon\big)\big) = 0.35 && p\big(\operatorname{subst}\big(s_1, (12, L)\big)\big) = 0.35\\ &p\big(\operatorname{subst}\big(s_1, (222, L)\big)\big) = 0.15 && p\big(\operatorname{subst}\big(s_1, (12, L)(222, L)\big)\big) = 0.15\\ &p\big(\operatorname{adj}(a, \varepsilon, s_1, (12, L))\big) = 1 && p\big(\operatorname{adj}(a, \varepsilon, s_1, (222, L))\big) = 1. \end{aligned}$$

Some of the generated graph's hyperedges are displayed in Fig. 4.6 and Fig. 4.7. We can use these edges to construct a derivation which describes the *when and where* of the substitution and adjoining operations employed to construct a certain derived tree. For instance, the derivation

$$\operatorname{subst}\big(s_1, (12, L)\big)\Big(\operatorname{subst}\big(s_2, \varepsilon\big)(), \operatorname{adj}\big(a, \varepsilon, s_1, (12, L)\big)()\Big) \in D^{\mathrm{S}}_{\mathrm{HG}(\mathscr{G})}$$

delineates the construction of a derived tree from our earlier example in Fig. 4.5. Step one from there, i.e. the step where we started out with the start non-terminal S, is not directly represented, but in the fact that the above derivation is one of S. The hyperedge $\mathrm{subst}(s_1, (12, L))$ denotes the substitution performed in step two, and moreover it activates the adjoining site $(12, L)$ for later adjoining. This adjunction, conducted in step three, is signified by $\mathrm{adj}(a, \varepsilon, s_1, (12, L))$, and the final substitution in step four with $\mathrm{subst}(s_2, \varepsilon)$, now with no further activated adjoining sites. Observe that the application order of the operations is not completely determined by the hypergraph derivation.

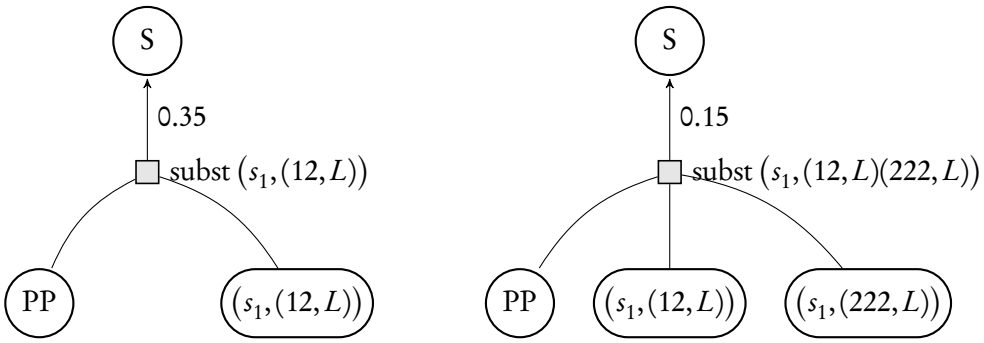


Figure 4.6: Two substitution hyperedges from Ex. 4.5

Figure 4.7: Adjoining hyperedges from Ex. 4.5

As stated above, the derivations of $\mathrm{HG}(\mathscr{G})$ only describe what substitution and adjoining operations are to be applied when and where—they do not immediately constitute the derived trees! However, we can give a function parse, which "executes" the operations contained in a hypergraph-derivation and returns the desired parse tree.

Since the foot nodes of auxiliary trees might change after them being adjoined into, we also introduce a helper function parse' which updates and returns the modified foot nodes accordingly.

Definition 4.6. Suppose a ptig $\mathscr{G} = (N, T, S, \mathscr{S}, \mathscr{A}, P)$, then we define

$$\begin{aligned}
&\mathrm{parse}\colon D^{\mathrm{subst}}_{\mathrm{HG}(\mathscr{G})} \to U_N(T)\\
&\mathrm{parse}\Big(\mathrm{subst}\big((\zeta, W, P^r_{\mathrm{adj}}), (w_1,\rho_1)\cdots(w_k,\rho_k)\big)\big(\Phi_1,\ldots,\Phi_n,\Psi_1,\ldots,\Psi_k\big)\Big)\\
&\quad = \chi_k\big(\zeta[\mathrm{parse}(\Phi_1),\ldots,\mathrm{parse}(\Phi_n)]_\lambda, w_1\cdots w_k, \xi_1\cdots\xi_k, *_1\cdots *_k\big)\\
&\qquad \text{where for every } i\in[k]\colon (\xi_i, *_i) = \mathrm{parse}'(\Psi_i)
\end{aligned}$$

with $n = |\mathrm{yield}_N(\zeta)|$ and $\lambda = v_1\cdots v_n$ the list of leaf positions from $\mathrm{lv}_N(\zeta) = \{v_1,\ldots v_n\}$, ordered by the lexicographic order on $\mathbb{N}^*$, i.e. $v_1 <_* v_2 <_* \cdots <_* v_n$. Moreover we define

$$\begin{aligned}
&\mathrm{parse}'\colon D^{\mathrm{adj}}_{\mathrm{HG}(\mathscr{G})} \to U_N(T)\times\mathbb{N}^*\\
&\mathrm{parse}'\Big(\mathrm{adj}\big((\zeta, W, *, P^r_{\mathrm{adj}}), (w_1,\rho_1)\cdots(w_k,\rho_k), s, v\big)\big(\Phi_1,\ldots,\Phi_m,\Psi_1,\ldots,\Psi_k\big)\Big)\\
&\quad = \chi'_k\big(\zeta[\mathrm{parse}(\Phi_1),\ldots,\mathrm{parse}(\Phi_m)]_\lambda, *, w_1\cdots w_k, \xi_1\cdots\xi_k, *_1\cdots *_k\big)\\
&\qquad \text{where for every } i\in[k]\colon (\xi_i, *_i) = \mathrm{parse}'(\Psi_i)
\end{aligned}$$

with $m = |\operatorname{yield}'_N(\zeta, *)|$ and $\lambda = v_1 \cdots v_m$ the list of non-foot leaf positions from $\operatorname{lv}_N(\zeta) \setminus \{*\} = \{v_1, \ldots v_m\}$, again ordered by the lexicographic order on $\mathbb{N}^*$. Here, the function domains $D^{\text{subst}}_{\text{HG}(\mathscr{G})}$ and $D^{\text{adj}}_{\text{HG}(\mathscr{G})}$ just signify that parse is to be applied to derivations with a subst-edge as root, and parse$'$ to those with an adj-edge:

$$D^{\text{subst}}_{\text{HG}(\mathscr{G})} = \{d \in D_{\text{HG}(\mathscr{G})} \mid d(\varepsilon) = \text{subst}(x, y) \text{ for some } x, y\}$$
$$D^{\text{adj}}_{\text{HG}(\mathscr{G})} = \{d \in D_{\text{HG}(\mathscr{G})} \mid d(\varepsilon) = \text{adj}(v, w, x, y) \text{ for some } v, w, x, y\}$$

The families of functions $\chi = (\chi_k \mid k \in \mathbb{N})$ and $\chi' = (\chi'_k \mid k \in \mathbb{N})$ do the brunt of the work for adjoining, accomplishing in parallel all of the adjoining operations into the sites specified in the according hyperedge, χ for adjoining into initial trees and χ' for adjoining into auxiliary trees. Again, χ' has to manage potential modifications of the foot node of the auxiliary tree adjoined into.

Definition 4.7. For every $k \in \mathbb{N}$, we define the functions

$$\begin{aligned}
&\chi_k \colon U_N(T) \times (\mathbb{N}^*)^k \times (U_N(T))^k \times (\mathbb{N}^*)^k \to U_N(T) \\
&\chi_0(\zeta, \varepsilon, \varepsilon, \varepsilon) = \zeta \\
&\chi_k(\zeta, w_1 \cdots w_k, \xi_1 \cdots \xi_k, *_1 \cdots *_k) \\
&\quad = \chi_{k-1}\big(\zeta[\xi_k[\zeta|_{w_k}]_{*_k}]_{w_k}, w_1 \cdots w_{k-1}, \xi_1 \cdots \xi_{k-1}, *_1 \cdots *_{k-1}\big) \qquad (k \geq 1)
\end{aligned}$$

as well as

$$\begin{aligned}
&\chi'_k \colon U_N(T) \times \mathbb{N}^* \times (\mathbb{N}^*)^k \times (U_N(T))^k \times (\mathbb{N}^*)^k \to U_N(T) \times \mathbb{N}^* \\
&\chi'_0(\zeta, *, \varepsilon, \varepsilon, \varepsilon) = (\zeta, *) \\
&\chi'_k(\zeta, *, w_1 \cdots w_k, \xi_1 \cdots \xi_k, *_1 \cdots *_k) \qquad (k \geq 1) \\
&\quad = \chi'_{k-1}\big(\zeta[\xi_k[\zeta|_{w_k}]_{*_k}]_{w_k}, *', w_1 \cdots w_{k-1}, \xi_1 \cdots \xi_{k-1}, *_1 \cdots *_{k-1}\big) \\
&\qquad \text{where } *' = \begin{cases} w_k *_k x & \text{for } * = w_k x \\ * & \text{otherwise.} \end{cases}
\end{aligned}$$

The arguments of a function χ_k, $k \in \mathbb{N}$, are the tree to adjoin into, and lists (each of length k) of adjoining site positions, of trees which are to be adjoined and of their respective foot nodes. A function χ'_k has, in addition to those, also an argument which accepts the supplied auxiliary tree's foot node. Each χ_k (or χ'_k) performs only one adjoining operation and leaves the rest of the work to χ_{k-1}, χ_{k-2}, … (resp. χ'_{k-1}, χ'_{k-2}, …).

Note that the successive substitutions performed by the χ_k and χ'_k do not interfere with each other because of the order of the adjoining positions $w_1, \ldots, w_k$: adjunction is only performed on a node after all adjoining operations in the subtree dominated by this node are completed.

Example 4.8. Let us continue Ex. 4.5 and show how the hypergraph derivation

$$\text{subst}\,(s_1, (12, L))\Big(\text{subst}\,(s_2, \varepsilon)\,(), \text{adj}\,(a, \varepsilon, s_1, (12, L))\,()\Big)$$

is turned into a derived tree. We have

$$\text{parse}\,\Big(\text{subst}\,(s_1, (12, L))\Big(\text{subst}\,(s_2, \varepsilon)\,(), \text{adj}\,(a, \varepsilon, s_1, (12, L))\,()\Big)\Big)$$

$$= \chi_1(\zeta_1[\,\text{parse}(\text{subst}(s_2,\varepsilon)())\,]_{223}, 12, \phi, *) \tag{4.1}$$

with

$$\begin{aligned}(\phi, *) &= \text{parse}'\,(\,\text{adj}(a,\varepsilon,s_1,(12,L))())\\ &= \chi_0'(\xi,2,\varepsilon,\varepsilon,\varepsilon)\\ &= (\xi,2),\end{aligned}$$

so we can rewrite (4.1) as

$$\begin{aligned}&\chi_1(\zeta_1[\,\text{parse}(\text{subst}(s_2,\varepsilon)())\,]_{223}, 12, \xi, 2)\\ &\quad= \chi_1(\zeta_1[\,\chi_0(\zeta_2,\varepsilon,\varepsilon,\varepsilon)\,]_{223}, 12, \xi, 2)\\ &\quad= \chi_1(\zeta_1[\zeta_2]_{223}, 12, \xi, 2)\\ &\quad= \chi_0(\zeta[\xi[\zeta|_{12}]_2]_{12},\varepsilon,\varepsilon,\varepsilon) \qquad \text{(with } \zeta = \zeta_1[\zeta_2]_{223}\text{)}\\ &\quad= \zeta[\xi[\zeta|_{12}]_2]_{12}.\end{aligned}$$

The value of $\zeta = \zeta_1[\zeta_2]_{223}$ is

$$\begin{aligned}\zeta &= \text{S}\Big(\text{NP}\big(\text{DT(The)}, \text{NN(dog)}\big), \text{VP}\big(\text{VBZ(bites)}, \text{NP}(\text{DT(the)}, \text{NN(man)}), \text{PP}\big)\Big)\\ &\qquad \Big[\text{PP}\big(\text{IN(in)}, \text{PRP\$(his)}, \text{NN(leg)}\big)\Big]_{223}\\ &= \text{S}\Big(\text{NP}\big(\text{DT(The)}, \text{NN(dog)}\big), \text{VP}\big(\text{VBZ(bites)}, \text{NP}(\text{DT(the)}, \text{NN(man)}),\\ &\qquad \text{PP}(\text{IN(in)}, \text{PRP\$(his)}, \text{NN(leg)})\big)\Big),\end{aligned}$$

so we obtain

$$\begin{aligned}\zeta|_{12} &= \text{NN(dog)},\\ \xi[\zeta|_{12}]_2 &= \text{NN}\big(\text{JJ(angry)}, \text{NN(dog)}\big),\end{aligned}$$

and finally,

$$\begin{aligned}\zeta[\xi[\zeta|_{12}]_2]_{12} = \text{S}\Big(&\text{NP}\big(\text{DT(The)}, \text{NN}\big(\text{JJ(angry)}, \text{NN(dog)}\big)\big),\\ &\text{VP}\big(\text{VBZ(bites)}, \text{NP}(\text{DT(the)}, \text{NN(man)}),\\ &\text{PP}(\text{IN(in)}, \text{PRP\$(his)}, \text{NN(leg)})\big)\Big).\end{aligned}$$

One can easily verify that this result corresponds to the derived tree at the bottom of Fig. 4.5.

4.2 Training with the State-Split Algorithm

As we mentioned above, ptigs, in contrast to probabilistic context-free grammars, can not only match one branch of a parse tree with a production rule, but several ones, enabling linguists to assign higher probabilities to certain common phrase structures. Because of this, they can be considered to be stochastically more powerful than pcfgs.

This higher expressivity comes at a price, however: For pcfgs, given a corpus of parse trees, there is a simple method for generating the *treebank grammar*, whose rule probabilities constitute a maximum-likelihood estimate for the supplied corpus, by reading off the used rules into the grammar and deriving the probability of each production rule from its relative frequency in the corpus, as more thoroughly described by Prescher (2005).

Since for ptigs we do not have such a simple correspondence between grammar rules and branches in a parse tree, we expect the methods to read them off to be somewhat more intricate. To simplify matters, we will not try to produce such a treebank grammar by ourselves, but presuppose that we already have a ptig G which more or less grasps the structure of the underlying corpus. This grammar may, e.g., have been generated by hand, or from an automated extraction technique like the one described by Chen and Shanker (2004).

However, we can still hope to further refine the grammar by supplying its non-terminal symbols with *annotations*. These annotations help model the phrasal structure of a supplied treebank corpus by introducing several roles for non-terminal symbols. The non-terminal for verb phrases, VP, may, e.g., be extended with a certain annotation for where it subsumes a transitive verb (i.e. a verb with an object) and another one, where it expands to an intransitive verb or a verb in infinitive form.

Such annotations might also be added manually by a linguist (compare the hand-generated XTAG grammar by Doran, Egedi, Hockey, Srinivas, and Zaidel, 1994), however the State-Split algorithm, first described by Petrov et al. (2006), allows this annotation process to be automated. In each iteration of this algorithm, the present annotation symbols are split in two, while the corresponding edge probabilities are normalized.

Afterwards, the Expectation-Maximization (see Prescher, 2005; Dempster et al., 1977), resp. the Inside-Outside algorithm (see Lari and Young, 1991; Manning and Schütze, 1999, p. 398), is applied, both facilitating maximum-likelihood estimates for probabilistic hypergraphs on incomplete data. This allows learning probabilities for the newly split edges which model the supplied corpora optimally in their statistical properties.

In this case, the incomplete data we proceed from consists of a corpus of annotated derivations from the previous State-Split step, while the complete data is given by all corresponding annotated derivations in the newly-split hypergraph. EM or the Inside-Outside algorithm then allow us to infer probabilities for annotations that maximize the likelihood of the corpus.

As mentioned by Prescher (2005), the Inside-Outside algorithm can be viewed as a dynamic programming instance of a more general EM algorithm. Since its approach may be easier to grasp, we will first give a definition of the State-Split procedure using EM, and then give an argument how the Inside-Outside algorithm can be seen as instantiating it in the case at hand.

Figure 4.8 shows the general idea of the State-Split algorithm: We start out with some treebank, i.e. with a corpus containing incomplete data in the form of full-parse trees, and with a grammar (in our case, this might be a ptig) whose likelihood is to be maximized for the supplied corpus.

First of all, the grammar is transformed into a hypergraph representation, as given in Sec-

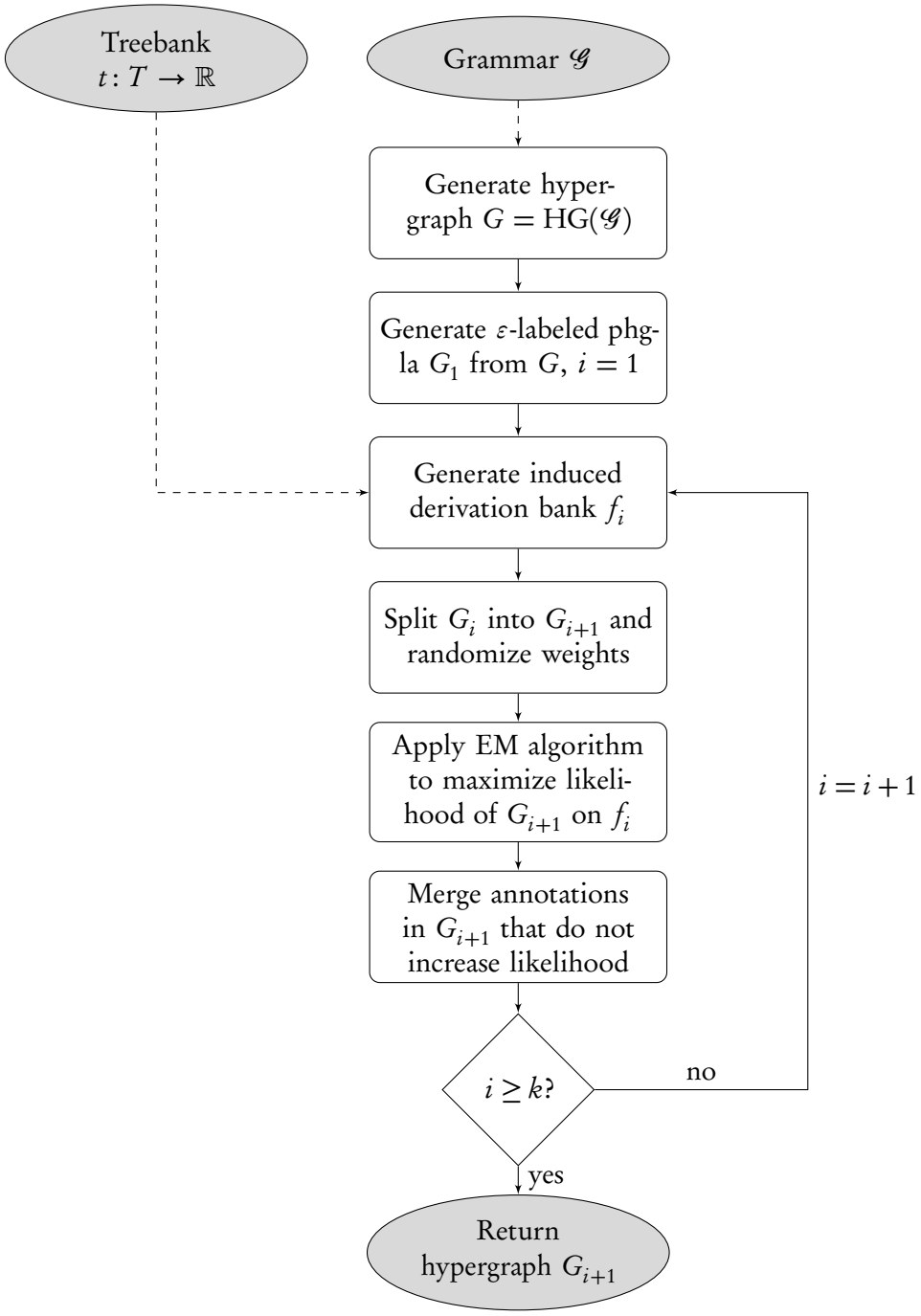


Figure 4.8: Flow chart for the State-Split procedure

tion 4.1 for ptigs. One should note that we will not revert this transformation: from now on, we will only concern ourselves with probabilistic hypergraphs. Hence, the following description of State-Split might also be applied to another grammar formalism, provided there is a way to represent it by a probabilistic hypergraph. In this way, we can construe hypergraphs as *interfaces,* which hide the concrete grammar formalism and allow unified access by this abstraction.

In the next step, the resulting phg G is augmented with a singleton set of annotation symbols $H = \{\varepsilon\}$, eventuating in an annotated hypergraph G_1. Note this step is in so far technical as it does not change the set of derivations of the hypergraph (up to isomorphism). However, it allows us to make an abstraction in the following step, expecting some arbitrary annotated hypergraph. This isomorphism, together with a parsing function mapping derivations to full-parse trees, is also employed to generate a *derivation bank* f_1, i.e. a corpus of derivations in the phg-la G_1, from the supplied treebank.

The next three steps comprise the actual brunt work of the State-Split procedure: We split each annotation symbol of the supplied hypergraph G_i into two subsymbols, adjusting the definitions of its edges and their probabilities, with the resulting graph G_{i+1}. Then the EM algorithm is applied, generating edge probabilities for G_{i+1} that maximize its likelihood on the incomplete-data corpus f_i.

Finally, we merge back together annotations which do not increase the graph's likelihood over a given threshold. On the one hand, this may constrain the exponential explosion of annotation symbols caused by doubling their number in each step, on the other hand, it prevents *oversplitting*, i.e. the case that phrasal categories are split without any linguistical justification for this. We repeat the steps described above a preconfigured number of times and then return the resulting split hypergraph.

4.2.1 Hypergraphs with latent annotations

In our endeavor to describe the State-Split algorithm, first of all, we will have to give a formal notion of how to add annotations to our language models. The annotation is thereby performed directly on the hypergraphs generated from ptigs, as described in Section 4.1 above. We hope this leads to a more lucid and general formulation of the idea. So, to begin with, let us define what we mean by *hypergraphs with latent annotations*.

Definition 4.9. Let H be a finite non-empty set. For a phg $G = (N, E, \mu, g, p)$, we call

- $N\langle H\rangle = N \times H$ the set of *complete nodes* of G with annotations from H and
- $E\langle H\rangle = \{(e, h) \mid e \in E, h = (x_1 \cdots x_k, x_0) \in H^k \times H, k = \mathrm{ar}(e)\}$ the set of *complete hyperedges* of G with annotations from H.

For a complete node $(a, x) \in N\langle H\rangle$, we will write $a\langle x\rangle$ and analogously $e\langle h\rangle$ for a complete edge $(e, h) \in E\langle H\rangle$. If we have $e\langle(x_1 \cdots x_k, x_0)\rangle \in E\langle H\rangle$, we will omit the parentheses and just write $e\langle x_1 \cdots x_k, x_0\rangle$ for better readability.

The concepts of heads and tails can be lifted to complete hyperedges as follows: assume $e\langle h\rangle \in E\langle H\rangle$ with $\mu(e) = (b_1 \cdots b_k, a)$ and $h = x_1 \cdots x_k, x_0$, then $\mathrm{hd}(e\langle h\rangle) = a\langle x_0\rangle$ and $\mathrm{tl}(e\langle h\rangle) = b_1\langle x_1\rangle \cdots b_k\langle x_k\rangle$. We also define $\mu(e\langle h\rangle) = (\mathrm{tl}(e\langle h\rangle), \mathrm{hd}(e\langle h\rangle))$.

Definition 4.10. A *probabilistic hypergraph with latent annotations (phg-la)* is a tuple $G = (N, E, H, \mu, g, p)$ where

- H is a finite non-empty set, called the set of *latent annotation symbols*,
- the sets of nodes N, edges E and the function $\mu: E \to N^* \times N$ are as in Def. 2.13 for probabilistic hypergraphs,
- $g \in N\langle H \rangle$ is called the *complete goal node* and
- $p: E\langle H \rangle \to [0,1]$ denotes *complete hyperedge probabilities.*

The class of all phg-las with annotations from a set H will be denoted by PHGLA_H.

As we can see, a phg-la does not really differ that much from a phg—it can actually be understood as a probabilistic hypergraph parameterized over H, giving rise to more or less subdivisions of every node and hyperedge. Because of this, many function definitions for phgs carry over naturally to phg-las. We will not redefine all these functions, instead a transformation γ from phg-las to phgs shall be given, which will be assumed to be implicitly invoked if such a function is applied to a phg-la.

Definition 4.11. For every phg-la $G = (N, E, H, \mu, g, p)$, its *induced hypergraph* is defined as $\gamma(G) = (N\langle H \rangle, E\langle H \rangle, \mu', g, p)$ with $\mu'(e\langle x_1 \cdots x_k, x_0 \rangle) = (b_1\langle x_1 \rangle \cdots b_k\langle x_k \rangle, a\langle x_0 \rangle)$, where $\mu(e) = (b_1 \cdots b_k, a)$.

So, for any set A, function $f: \mathrm{PHG} \to A$, and phg-la G, we will introduce the abbreviaton $f(G) = f(\gamma(G))$. We will also define how to annotate derivations. These will serve as complete data later on.

Definition 4.12. Assuming a phg-la $G = (N, E, H, \mu, g, p)$, and a complete node $a\langle x \rangle \in N\langle H \rangle$, we define the set $A_G^{a\langle x \rangle}$ of *annotated (or complete) derivations* of this node as the smallest set A with

$$A = \{e\langle h \rangle(d_1, \ldots, d_k) \mid e\langle h \rangle \in E\langle H \rangle, \mu(e\langle h \rangle) = (b_1\langle x_1 \rangle \cdots b_k\langle x_k \rangle, a\langle x_0 \rangle), \\ d_i \in A_G^{b_i\langle x_i \rangle} \text{ for } i \in [k]\}.$$

Again, we can define the set of all annotated derivations of G as $A_G = \bigcup_{a\langle x \rangle \in N\langle H \rangle} A_G^{a\langle x \rangle}$.[4] Note that via the definition of the induced hypergraph of a phg-la $G = (N, E, H, \mu, g, p)$ from Def. 4.12 above, there arises naturally a probability distribution on A_G^g, denoted again by $P(\cdot \mid G)$.

Let us introduce the notions of *treebanks* and *derivation banks,* which will serve as the corpora used in the learning of optimal edge probabilities later on. These depend on the definition of *full-parse trees,* which can be thought of as the derived trees of the grammar formalism underlying the respective hypergraph.

Definition 4.13. For a phg $G = (N, E, \mu, g, p)$, we call a countable set $T \subseteq U_\Sigma(A)$ (for some anonymous finite sets Σ, A) a set of *full-parse trees (for G)* if there is a function parse: $D_G^g \to T$, called the *parsing function.*[5]

[4] The definition of A_G for a phg-la G corresponds to the one for unannotated derivations, i.e. of $D_{G'}$ for a phg G'. The sole purpose behind the introduction of A_G is the distinction between annotated and unannotated derivations.

[5] Note that this is the case for the instance of phgs derived from ptigs, as defined in the previous Section 4.1, together with the suggestively named function parse, restricted to the domain D_G^g.

Throughout the rest of this work, we will assume T to be a (no further specified) set of full-parse trees for the hypergraph G (if mentioned at all), and parse *the associated function as above.*

Definition 4.14. A corpus $f : T \to \mathbb{R}$ of full-parse trees is called a *treebank* if it is finite and non-empty. Similarly, presuming a phg-la G with goal node g, we call a corpus $f : A_G^g \to \mathbb{R}$ of annotated derivations in G a *derivation bank for* G, again under the requirement that it is finite and non-empty.

4.2.2 Splitting and Merging

With the above definition for a phg-la, let us derive how to accomplish the steps of initialization, splitting and merging displayed in Fig. 4.8. First of all, how can we transform a phg into a phg-la with identical derivations (up to isomorphism)?

Definition 4.15. The function init lifts some phg into a phg-la with the empty word as the only latent annotation symbol.

$$\begin{aligned}&\text{init}\colon \text{PHG} \to \text{PHGLA}_{\{\varepsilon\}}\\&\text{init}(N_1, E_1, \mu_1, g_1, p_1) = (N_2, E_2, \{\varepsilon\}, \mu_2, g_2, p_2)\end{aligned}$$

where

- $N_2 = N_1$, $E_2 = E_1$ and $\mu_2 = \mu_1$,
- $g_2 = g_1\langle\varepsilon\rangle$ and
- $p_2(e\langle h\rangle) = p_1(e)$ (since we must have $h = (\varepsilon^{|\text{tl}(e)|}, \varepsilon)$).

Next, presuming a phg-la with annotations from H, we can split those into $H \cdot \{0, 1\}$. This is modeled by the function split. It distributes the probability mass of each hyperedge evenly to its split edges and, by normalizing these weights, makes sure that the resulting phg-la is proper.

Definition 4.16. Assume a set of latent annotation symbols H. We define the function

$$\begin{aligned}&\text{split}\colon \text{PHGLA}_H \to \text{PHGLA}_{H\cdot\{0,1\}}\\&\text{split}(N_1, E_1, H, \mu_1, g_1, p_1) = (N_2, E_2, H \cdot \{0, 1\}, \mu_2, g_2, p_2)\end{aligned}$$

where

- $N_2 = N_1$, $E_2 = E_1$ and $\mu_2 = \mu_1$,
- $g_2 = a\langle x0\rangle$ where $g_1 = a\langle x\rangle$ and
- $p_2(e\langle h_1x_1 \cdots h_kx_k, h_0x_0\rangle) = \dfrac{p_1(e\langle h_1 \cdots h_k, h_0\rangle)}{2^k}$ for every $e\langle h_1x_1 \cdots h_kx_k, h_0x_0\rangle \in E_2\langle H_2\rangle$, where $h_i \in H_1$ and $x_i \in \{0, 1\}$ for $i \in \{0, \ldots, k\}$.

As stated above, split preserves properness of its argument: we have, for $x_0 \in \{0,1\}$ and $e \in E$,

$$\begin{aligned}
&\sum_{x_1,\dots,x_k \in \{0,1\}} p_2(e\langle h_1 x_1 \cdots h_k x_k, h_0 x_0\rangle) \\
&\quad = \sum_{x_1,\dots,x_k \in \{0,1\}} \frac{p_1(e\langle h_1 \cdots h_k, h_0\rangle)}{2^k} \\
&\quad = 2^k \cdot \frac{p_1(e\langle h_1 \cdots h_k, h_0\rangle)}{2^k} \\
&\quad = p_1(e\langle h_1 \cdots h_k, h_0\rangle),
\end{aligned}$$

and hence, for every complete node $a\langle xi\rangle$,

$$\sum_{\substack{e\langle h\rangle \in E\langle H\cdot\{0,1\}\rangle \\ \mathrm{hd}(e\langle h\rangle)=a\langle xi\rangle}} p_2(e\langle h\rangle) = \sum_{\substack{e\langle h\rangle \in E\langle H\rangle \\ \mathrm{hd}(e\langle h\rangle)=a\langle x\rangle}} p_1(e\langle h\rangle)$$

This sums up to one if the input hypergraph is already proper. Moreover, we need a function to merge together complete nodes that do not increase likelihood. Say we want to merge back together the two complete nodes $a\langle x0\rangle$ and $a\langle x1\rangle$. In order to accomplish that, the function $\mathrm{merge}_{a\langle x\rangle}$ sets the probabilities of hyperedges containing $a\langle x1\rangle$ to zero. Those hyperedges which only contain $a\langle x0\rangle$ then receive the additional probability mass, so that the result is again proper.

Definition 4.17. Assume a phg-la $G = (N, E, H, \mu, g, p)$ and let $a\langle x0\rangle \in N\langle H\rangle$ and $a\langle x1\rangle \in N\langle H\rangle$ be the complete nodes that are to be merged. Then we define

$$\begin{aligned}
&\mathrm{merge}^a_x : \mathrm{PHGLA}_H \to \mathrm{PHGLA}_H \\
&\mathrm{merge}^a_x(N_1, E_1, H, \mu_1, g_1, p_1) = (N_2, E_2, H, \mu_2, g_2, p_2)
\end{aligned}$$

where

- again, $N_2 = N_1, E_2 = E_1, \mu_2 = \mu_1$, as well as $g_2 = g_1$ and
- for every $e \in E$ with $\mu(e) = (a_1 \cdots a_k, a_0)$,

$$p_2(e\langle x_1 \cdots x_k, x_0\rangle) = \begin{cases} 0 & \text{if there is an } i \in \{0,\dots,k\} \\ & \text{with } a_i = a \text{ and } x_i = x1 \\ \displaystyle\sum_{h' \in \mathrm{subs}_{a\langle x\rangle}(e\langle x_1 \cdots x_k, x_0\rangle)} \frac{p_1(e\langle h'\rangle)}{2} & \text{if } a_0 = a \text{ and } x_0 = x0 \\ & \text{and there is no } i \in \{0,\dots,k\} \\ & \text{with } a_i = a \text{ and } x_i = x1 \\ \displaystyle\sum_{h' \in \mathrm{subs}_{a\langle x\rangle}(e\langle x_1 \cdots x_k, x_0\rangle)} p_1(e\langle h'\rangle) & \text{otherwise.} \end{cases}$$

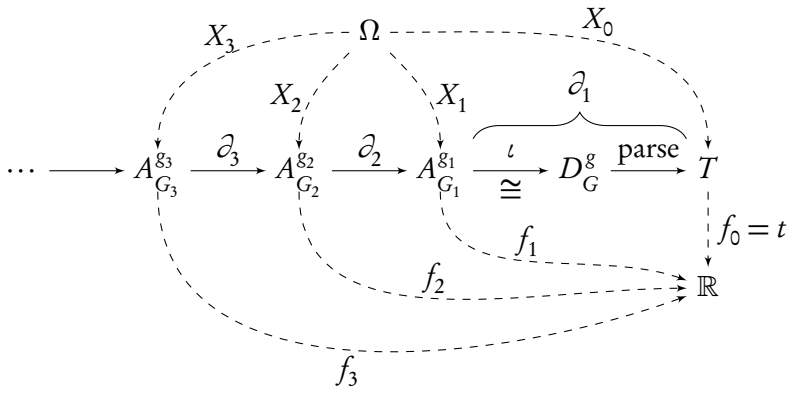


Figure 4.9: Functions, random elements and corpora appearing during State-Split

The set $\mathrm{subs}_{a\langle x\rangle}(e\langle h\rangle)$ is defined for every $e \in E$ with $\mu(e) = (a_1 \cdots a_k, a_0)$ by

$$\begin{aligned}\mathrm{subs}_{a\langle x\rangle}(e\langle x_1 \cdots x_k, x_0\rangle) = \{(x'_1 \cdots x'_k, x'_0) \mid &\text{for every } i \in \{0,\ldots,k\}, \text{ if } x_i = x0 \text{ and } a_i = a,\\ &\text{then } x'_i \in \{x\} \cdot \{0,1\}, \text{ else } x'_i = x_i\}.\end{aligned}$$

The rather complicated case distinction above accomplishes the following: all hyperedges containing the complete node $a\langle x1\rangle$ are set to probability zero (and might be pruned later), while edges containing $a\langle x0\rangle$ receive the additional probability mass of their counterparts with $a\langle x1\rangle$ instead of $a\langle x0\rangle$. In the case that this complete node appears in the hyperedge's head, we have to normalize the resulting probability by dividing by two. Again, this is in order to preserve properness of the hypergraph.

Because the probabilities of complete hyperedges are evenly distributed after splitting, we will introduce a function to randomize the edge weights of a phg-la. This breaks the symmetry and provides the EM algorithm with some kind of direction to develop the hyperedge probabilities into. The maximum degree of the introduced randomness is set via a parameter θ.

Definition 4.18. For a given phg-la G, the function randomize returns the set of all phg-las with edge probabilities close to the ones of G.

$$\begin{aligned}&\text{randomize}\colon \mathrm{PHGLA}_H \times \mathbb{R} \to 2^{\mathrm{PHGLA}_H}\\ &\text{randomize}(G,\theta) = \{\, G' \mid G' = (N,E,H,\mu,g,p'),\\ &\qquad\qquad G = (N,E,H,\mu,g,p),\\ &\qquad\qquad 0 < \left|p'(e\langle h\rangle) - p(e\langle h\rangle)\right| < \theta \text{ for every } e\langle h\rangle \in E\langle H\rangle,\\ &\qquad\qquad G' \text{ is consistent and proper}\}\end{aligned}$$

In order to give a concise mathematical characterization of the State-Split procedure, we define several more functions, compare Fig. 4.9.[6] Let, for some initial phg $G = (N,E,H_i,\mu,g,p)$, $G_1 = \mathrm{init}(G)$ be the phg-la which is effectively G, but with ε as only latent annotation symbol. Assume G_2, G_3, ..., are the phg-las produced at the end of each separate State-Split step. Their components are designated as $G_i = (N,E,H_i,\mu,g_i,p_i)$ for $i \geq 1$.

[6]Note that this diagram is not entirely commutative, e.g., we will certainly not have $f_i = f_{i-1} \circ \partial_i$ in every case.

Definition 4.19. For these G_i's derivation sets, we define a sequence of *partial strip* functions $\partial = (\partial_i \mid i \in \mathbb{N}^+)$ by

$$\partial_1 : A_{G_1} \to T$$
$$\partial_1 = \text{parse} \circ \iota,$$

where ι is the obvious isomorphism between A_{G_1} and D_G, $G_1 = \text{init}(G)$. For $j > 1$, we first of all define a function $\widetilde{\partial}_j$ on complete hyperedges:

$$\widetilde{\partial}_j : E\langle H_j \rangle \to E\langle H_{j-1} \rangle$$
$$\widetilde{\partial}_j(e\langle x_1 b_1 \cdots x_k b_k, x_0 b_0 \rangle) = e\langle x_1 \cdots x_k, x_0 \rangle$$

with $b_0, \ldots, b_k \in \{0, 1\}$. The function ∂_j is then the homomorphic lift of $\widetilde{\partial}_j$ to the set of complete derivations:

$$\partial_j : A_{G_j} \to A_{G_{j-1}}$$
$$\partial_j(e\langle h \rangle(d_1, \ldots, d_k)) = \widetilde{\partial}_j(e\langle h \rangle)(\partial_j(d_1), \ldots, \partial_j(d_k)).$$

As the latter function is just a lifted version of the former, we will henceforth omit the tilde from $\widetilde{\partial}_j$ and write ∂_j for both functions.[7]

The functions ∂_i, $i \geq 1$, project annotated derivations from a level of the split hierarchy to the less annotated derivations in the level below they originated from, i.e., they strip off a level of annotation. Thus, their inverses ∂_i^{-1} act as *symbolic analyzers* (as defined by Prescher, 2005) and will allow EM to distribute the probability masses of less-split derivations to their annotated version.

Moreover, we assume a sequence of random elements $X_0, X_1, \ldots$ with $X_0 : \Omega \to T$ and $X_j : \Omega \to A_{G_j}^{g_j}$ for $j > 0$ and demand that, for every $j \geq 0$,

$$X_j = \partial_{j+1} \circ X_{j+1}.$$

Such a random element X_i will later allow quantification over annotated derivations created in the i-th State-Split step. In the diagram, the functions $f_i : A_{G_i} \to \mathbb{R}$, $i \geq 1$ are derivation banks on the sets of annotated derivations of the phg-las G_i, subsequently generated from their predecessors in the State-Split procedure.

With these definitions, we can now display the State-Split procedure in pseudocode, as shown in Alg. 4.1. In every iteration of State-Split's main loop, we will have to compute a derivation bank for the newly-split hypergraph, which will be used as input to the invocation of EM (or Inside-Outside). It is computed from the derivation bank from the last step by distribution of its values according to the probability distribution on complete derivations, conditioned on

[7] Mind that in Fig. 4.9, the domains and codomains of the functions ∂_i, $i > 1$, have been implicitly restricted to derivations of the goal nodes g_i. This is due to the fact that the derivation banks f_i have these sets as their domains.

the probabilities of their less-annotated originators.[8] In the following, an iteration of the outer while-loop of the State-Split algorithm will also be called a *split-merge cycle*.

The algorithm has as parameters a phg G and a treebank t on which State-Split is to be performed. The integer k denotes the number of split-merge cycles intended, while the real number θ gives a measure of how much to randomize edge probabilities in each step, and η is passed on to EM (resp. Inside-Outside), indicating how much it should try to increase the likelihood. Finally, κ serves as a lower limit for the decrease in likelihood introduced by merging two complete nodes: if the ratio between the likelihood after and before merging drops below κ, the merge is not carried through, else, it is. For a small example run of State-Split, you may refer to Chapter 5.

Algorithm 4.1 The State-Split procedure for phg-las

Input: A phg $G=(N,E,\mu,g,p)$, a treebank $t: T \to \mathbb{R}$ and values $k \in \mathbb{N}$, $\theta, \eta \in \mathbb{R}$, $\kappa \in [0,1[$
Output: **StateSplit**$(G,t,k,\theta,\eta,\kappa)$

$f_0 = t$
$i \leftarrow 1$
$G_1 \leftarrow \text{init}(G)$ *// Add the empty annotation symbol*
$H_1 \leftarrow \{\varepsilon\}$
while $i \leq k$ **do**
  Compute $f_i : A_G \to \mathbb{R}$ acc. to

$$f_i(d) = \begin{cases} f_{i-1}(d) \cdot P(X_i = d \mid X_{i-1} = \partial_i(d), G_i) & \text{if } P(X_{i-1} = \partial_i(d) \mid G_i) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

  $G' \leftarrow \text{split}(G_i)$ *// Split each complete node into two*
  $H_{i+1} \leftarrow H_i \cdot \{0,1\}$
  randomly choose one $G'' \in \text{randomize}(G', \theta)$
  $G_{i+1} \leftarrow \textbf{EM}(G'', f_i, \partial_{i+1}, X_{i+1}, X_i, \eta)$ *// Try to find mle via EM*
  // Alternatively: $G_i \leftarrow$ ***Inside-Outside***$(G'', f_i, \partial_{i+1}, X_{i+1}, X_i, \eta)$
  for all $a \in N$, $x \in H_i$ **do**
    if $L(f_i; P(\cdot \mid \text{merge}^a_x(G_{i+1}))) / L(f_i; P(\cdot \mid G_{i+1})) > \kappa$ **then**
      $G_{i+1} \leftarrow \text{merge}^a_x(G_{i+1})$ *// Merge together complete nodes with no benefit to likelihood*
    end if
  end for
  $i \leftarrow i+1$
end while
return G_i

[8] As the definition of complete-data derivation banks fits thematically better to EM, compare Def. 4.20 in the appropriate subsection.

4.2.3 The Expectation-Maximization Algorithm

As shown in Fig. 4.8, in each split-merge cycle we apply the *Expectation-Maximization (EM) algorithm*. This algorithm allows us to give a try at finding the parameters for the maximum-likelihood estimate of some probability model on unbeknown *complete data,* possessing only knowledge of *incomplete data,* which may, e.g., be comprised of partial information projected from the complete data.

Note that this problem is inherently self-referential: if we knew the model parameters for the maximum-likelihood estimate, it would be fairly easy to derive the unknown complete data. Conversely, the knowledge of the actual complete data would make it trivial to devise the according maximum-likelihood estimate. EM breaks this cyclic dependency by starting with some random guess for the mle, then it alternatingly computes estimates for the induced statistical distribution of the complete data (in the *E-step*) and turns these into new, statistically fitting, probability distributions (the *M-step*), hopefully giving rise to better approximations of the mle. The procedure stops the described loop when the step-to-step difference of the likelihoods (or alternatively, the model parameters) falls beneath a certain threshold.

One should keep in mind that EM does not actually promise to find the real mle in every instance, it may well be the case that the procedure stops at some local maximum of the likelihood function, or even just at one of its saddle points. However, Dempster et al. (1977) have proved that the sequences of generated probability distributions are monotonically nondecreasing in their likelihoods, and combined with the practice of letting the algorithm run several times with various initial guesses and taking the best generated guess, this suffices for most practical cases, as remarked by Gupta and Chen (2011, p. 224).

In the scenario at hand, the probability model $\mathscr{M}_G$, whose maximum-likelihood estimate is sought, is induced by the hypergraph G underlying the initial phg-la in question, and its parameters, which are to be optimized, are the corresponding edge probabilities. The complete data is thought to be a corpus of fully annotated derivations, while the incomplete data, which we can effectively observe in the input derivation (resp. tree) bank, comprises the same derivations, but with the last level of annotations removed (resp., for the first step of the State-Split algorithm, the full-parse trees in the treebank).

We will develop the notion of this algorithm in the following, trailed by the explanation of a more efficient version of EM using dynamic programming, called the *Inside-Outside algorithm.* For that, we will use the random elements X and Y, which, on the one hand, allow formally well-defined probabilities, and, on the other hand, make the development more succinct. Here, the random element $X : \Omega \to A_G^g$ signifies complete data, and $Y : \Omega \to \partial(A_G^g)$ with $Y = \partial \circ X$, allows us to express statements over incomplete data. Thereby, the function ∂ will map complete to incomplete data.

The random element X, as well as the function ∂, will be plugged in with $X_1, X_2, \ldots$, resp. $\partial_1, \partial_2, \ldots$—and therefore Y with $X_0, X_1, \ldots$—by the State-Split procedure, according to the progress of its execution.

Definition 4.20. Supplied with a phg-la G with goal node g, random elements $X : \Omega \to A_G^g$, $Y : \Omega \to \partial(A_G^g)$, the respective function $\partial : A_G^g \to \partial(A_G^g)$ stripping away the last level of annotation from derivations of G, and a derivation bank $f : \partial(A_G^g) \to \mathbb{R}$ of incompletely annotated derivations, we can compute the *complete derivation bank* f_G expected by G, given

by

$$f_G : A_G^g \to \mathbb{R}$$
$$f_G(d) = \begin{cases} f(\partial(d)) \cdot P(X = d \mid Y = \partial(d), G) & \text{if } P(Y = \partial(d) \mid G) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Note that essentially, this definition serves no other purpose than distributing f's type frequencies to the types of f_G, according to G's induced probability distribution on complete data, i.e. on X, conditioned on the incomplete data indexed by Y. In the EM algorithm, this computation will be called the *E-step (expectation step).* It deserves this name because f_G denotes our expectation of the complete corpus, given the incomplete corpus f and the parameters induced by G.

Moreover, we can describe what we mean by a *derivation bank hypergraph:* presuming some derivation bank f for a phg-la G, it is the hypergraph G, with new edge probabilities derived from their statistic frequencies in f.

Definition 4.21. Assuming a derivation bank $f : A_G^g \to \mathbb{R}$ for a phg-la $G = (N, E, H, \mu, g, p)$, the *derivation bank hypergraph* for f and G is defined by $G_f = (N, E, H, \mu, g, p_f)$ with

$$p_f(e\langle h \rangle) = \frac{\sum_{d \in A_G^g} f(d) \cdot \zeta_{e\langle h \rangle}(d)}{\sum_{d \in A_G^g} f(d) \cdot \zeta_{\mathrm{hd}(e\langle h \rangle)}(d)}, \tag{4.2}$$

assuming the fraction's denominator is not zero, else

$$p_f(e\langle h \rangle) = \frac{1}{|\{e'\langle h' \rangle \in E\langle H \rangle \mid \mathrm{hd}(e'\langle h' \rangle) = \mathrm{hd}(e\langle h \rangle)\}|}.$$

The assignment in the latter case is defined in this way to guarantee properness of the derivation bank hypergraph. The functions $\zeta_{e\langle h \rangle}$ and $\zeta_{a\langle x \rangle}$ hereby give the counts of appearance of the respective complete hyperedges or nodes in a derivation:

$$\zeta_{e\langle h \rangle}(d) = |\{w \in \mathrm{pos}(d) \mid d(w) = e\langle h \rangle\}|$$

for every complete hyperedge $e\langle h \rangle \in E\langle H \rangle$ and

$$\zeta_{a\langle x \rangle}(d) = |\{w \in \mathrm{pos}(d) \mid \mathrm{hd}(d(w)) = a\langle x \rangle\}|$$

for every complete node $a\langle x \rangle \in N\langle H \rangle$.

So the definition in (4.2) assigns every hyperedge as probability the ratio between the number of times they appeared in the derivation bank f (weighted by the values of f on the respective derivation) and the number of times they *could* have appeared there, i.e. the count of appearance of their head nodes (again weighted by f). The derivation hypergraph is proper and consistent (cf. Chi and Geman, 1998). It constitutes a maximum-likelihood estimate of G's induced probability model on the given derivation bank, compare the proof for treebanks and pcfgs by Prescher (2005). This fact justifies the name *M-step (Maximization-Step)* for this computation in the EM algorithm.

Algorithm 4.2 The Expectation-Maximization algorithm for phg-las

Input: A phg-la $G = (N, E, H, \mu, g, p)$, a corpus $f : A_G^g \to \mathbb{R}$, $\partial : A_G \to \partial(A_G)$, random elements $X : \Omega \to A_G^g$ and $Y : \Omega \to \partial(A_G^g)$ and a value $\theta \in \mathbb{R}$.
Output: $\mathbf{EM}(G, f, \partial, X, Y, \theta)$

$i \leftarrow 0$
$G_0 = (N, E, \mu, g, p_0) \leftarrow G$
repeat
$i \leftarrow i + 1$
E-step: Compute the derivation bank $f_i : A_G \to \mathbb{R}$ defined by

$$f_i(d) = \begin{cases} f(\partial(d)) \cdot P(X = d \mid Y = \partial(d), G_{i-1}) & \text{if } P(Y = \partial(d) \mid G_{i-1}) > 0 \\ 0 & \text{otherwise.} \end{cases} \quad \textit{// compare Def. 4.20}$$

M-step: $G_i \leftarrow (N, E, H, \mu, g, p_i)$ where for all $e\langle h \rangle \in E\langle H \rangle$
if $\sum_{d \in A_G^g} f_i(d) \cdot \zeta_{\mathrm{hd}(e\langle h \rangle)}(d) \neq 0$ **then**

$$p_i(e\langle h \rangle) = \frac{\sum_{d \in A_G^g} f_i(d) \cdot \zeta_{e\langle h \rangle}(d)}{\sum_{d \in A_G^g} f_i(d) \cdot \zeta_{\mathrm{hd}(e\langle h \rangle)}(d)} \quad \textit{// compare Def. 4.21}$$

else
$p_i(e\langle h \rangle) = 1/|\{e'\langle h' \rangle \in E\langle H \rangle \mid \mathrm{hd}(e'\langle h' \rangle) = \mathrm{hd}(e\langle h \rangle)\}|$
end if
until $L\left(f; P(\cdot \mid G_i)\right) - L\left(f; P(\cdot \mid G_{i-1})\right) < \theta$
return G_i

The EM algorithm can then be formulated as shown in Alg. 4.2. In each iteration of its outer loop it refines the edge probabilities of the supplied hypergraph by consecutively applying the E- and the M-step. As noted by Prescher (2005), the likelihood of the subsequently generated phg-la's probability distributions on the corpus is non-decreasing, i.e. we have

$$L(f; P(\cdot \mid G_0)) \leq L(f; P(\cdot \mid G_1)) \leq L(f; P(\cdot \mid G_2)) \leq \cdots$$

The algorithm is provided with a real-valued argument $\theta \in \mathbb{R}$. When the increase in likelihood between iterations falls beneath θ, the EM algorithm breaks and returns the lastly generated phg-la.

4.2.4 The Inside-Outside Algorithm

One should take note that the formulation of the EM procedure above does not promise the greatest performace in real world applications: after all, in the fraction in Def. 4.21, we take a sum over all annotated derivations $d \in A_G$. The problem in doing so lies not in the fact that A_G is infinite—after all, for a finite corpus f, we would just have to sum over those $d \in A_G$ with $f(d) > 0$. However, we would still have to sum over all possible annotated versions of each partially incomplete derivation $d \in \partial(A_G)$. The more the hypergraph is split, the more this summation carries into weight.

So it is clearly advisable to devise a means to get rid of the summation over completely annotated derivations. The *Inside-Outside algorithm* (Lari and Young, 1991) accomplishes that and can be seen as an instance of EM, as witnessed by the following equations, based on a similar derivation of the Forward-Backward algorithm for hidden Markov models by Gupta and Chen (2011).

Let us start out with the reestimation formula for the M-step as given in Alg. 4.2 (presuming that the fraction's denominator is not zero):

$$p_i(e\langle h\rangle) = \frac{\sum_{d \in A_G^g} f_i(d) \cdot \zeta_{e\langle h\rangle}(d)}{\sum_{d \in A_G^g} f_i(d) \cdot \zeta_{\mathrm{hd}(e\langle h\rangle)}(d)}. \tag{4.3}$$

We can rewrite the fraction's numerator as follows:[9]

$$\begin{aligned}
&\sum_{d \in A_G^g} f_i(d) \cdot \zeta_{e\langle h\rangle}(d) \\
&= \sum_{\substack{d \in A_G^g \\ P(Y=\partial(d)|G_{i-1})>0}} f(\partial(d)) \cdot P(X = d \mid Y = \partial(d), G_{i-1}) \cdot \zeta_{e\langle h\rangle}(d) && \text{(by Def. 4.20)} \\
&= \sum_{\substack{d' \in \partial(A_G^g) \\ P(Y=d'|G_{i-1})>0}} f(d') \cdot \sum_{d \in \partial^{-1}\{d'\}} P(X = d \mid Y = d', G_{i-1}) \cdot \zeta_{e\langle h\rangle}(d) && \text{(by distributivity)} \\
&= \sum_{\substack{d' \in \partial(A_G^g) \\ P(Y=d'|G_{i-1})>0}} f(d') \cdot \sum_{d \in \partial^{-1}\{d'\}} P(X = d \mid Y = d', G_{i-1}) \cdot \sum_{w \in \mathrm{pos}(d)} \delta(d(w), e\langle h\rangle) \\
&= \sum_{\substack{d' \in \partial(A_G^g) \\ P(Y=d'|G_{i-1})>0}} f(d') \cdot \sum_{w \in \mathrm{pos}(d')} \sum_{d \in \partial^{-1}\{d'\}} \left(P(X = d \mid Y = d', G_{i-1}) \cdot \delta(d(w), e\langle h\rangle)\right) \\
&&& \text{(since } \mathrm{pos}(d) = \mathrm{pos}(d')\text{)}
\end{aligned}$$

[9] We should formally describe what we mean by the notation $X(w)$, which mimics the access of a node $d(w)$ in a derivation d and should, in this case, *not* be interpreted as application of the function X to w, but rather as a new random element derived from the random element X. Define, for a phg-la $G = (N, E, H, \mu, g, p)$, and every $w \in \mathbb{N}^*$, a function $\nu_w : A_G \to E\langle H\rangle \cup \{\bot\}$ with

$$\nu_w(d) = \begin{cases} d(w) & \text{if } w \in \mathrm{pos}(d) \\ \bot & \text{otherwise,} \end{cases}$$

then $X(w) : \Omega \to E\langle H\rangle \cup \{\bot\}$ with $X(w) = \nu_w \circ X$. The definition of hd must then also be extended by the equation $\mathrm{hd}(\bot) = \bot$, to allow access to the heads of hyperedges in a derivation indexed by X. Then, $\mathrm{hd}(X(w))$ is also a random element, with the type $\mathrm{hd}(X(w)) : \Omega \to N\langle H\rangle \cup \{\bot\}$.

$$= \sum_{\substack{d' \in \partial(A_G^g) \\ P(Y=d' \mid G_{i-1})>0}} f(d') \cdot \sum_{w \in \mathrm{pos}(d')} \sum_{d \in \partial^{-1}\{d'\}} P(X=d, X(w)=e\langle h\rangle \mid Y=d', G_{i-1})$$

$$= \sum_{\substack{d' \in \partial(A_G^g) \\ P(Y=d' \mid G_{i-1})>0}} f(d') \cdot \sum_{w \in \mathrm{pos}(d')} P(X(w)=e\langle h\rangle \mid Y=d', G_{i-1}). \qquad \text{(by Lem. 2.9)}$$

By an analogous argument, we can reformulate the denominator of the above fraction as

$$\sum_{\substack{d' \in \partial(A_G^g) \\ P(Y=d' \mid G_{i-1})>0}} f(d') \cdot \sum_{w \in \mathrm{pos}(d')} P(\mathrm{hd}(X(w)) = \mathrm{hd}(e\langle h\rangle) \mid Y=d', G_{i-1}).$$

We managed to remove the summation over completely annotated derivations! And moreover, as the only used derivation bank is the incomplete f, coming from the last State-Split step, we can also do away with the tedious computation of complete corpora in every iteration of EM. But how to compute the probabilities in the second part? Let us define two recursive functions, computing the *inside* and *outside probabilities* of a derivation at a given position, these will aid us in this task.

Definition 4.22. Assume a phg-la $G = (N, E, H, \mu, g, p)$ as from above (i.e., $H = H' \cdot \{0,1\}$ for some H') and a derivation $d \in \partial(A_G)$. The *inside probability* $\beta_d^w(a\langle x\rangle)$ of a complete node $a\langle x\rangle \in N\langle H\rangle$ at a position $w \in \mathrm{pos}(d)$ is given by

$$\beta_d^w(a\langle x\rangle \mid G) = \beta'_{d|_w}(a\langle x\rangle \mid G)$$

$$\text{where } \beta'_{e'\langle h'\rangle(d_1,\ldots,d_k)}(a\langle x\rangle \mid G) = \sum_{\substack{e\langle h\rangle \in \partial^{-1}\{e'\langle h'\rangle\} \\ \mu(e\langle h\rangle)=(b_1\langle x_1\rangle \cdots b_k\langle x_k\rangle, a\langle x\rangle)}} \beta'_{d_1}(b_1\langle x_1\rangle \mid G) \cdots \beta'_{d_k}(b_k\langle x_k\rangle \mid G) \cdot p(e\langle h\rangle),$$

while the *outside probability* $\alpha_d^w(a\langle x\rangle \mid G)$ of $a\langle x\rangle \in N\langle H\rangle$ at $w \in \mathrm{pos}(d)$ can be computed as follows:

$$\alpha_d^\varepsilon(a\langle x\rangle \mid G) = \begin{cases} 1 & \text{if } d \in \partial\left(A_G^g\right) \text{ and } a\langle x\rangle = g \\ 0 & \text{otherwise} \end{cases}$$

$$\alpha^{wi}_{e'\langle h'\rangle(d_1,\ldots d_k)}(a\langle x\rangle \mid G) = \sum_{\substack{e\langle h\rangle \in \partial^{-1}\{e'\langle h'\rangle\} \\ \mu(e\langle h\rangle)=(b_1\langle x_1\rangle \cdots b_k\langle x_k\rangle, b_0\langle x_0\rangle) \\ b_i\langle x_i\rangle = a\langle x\rangle}} \alpha_d^w(b_0\langle x_0\rangle \mid G) \cdot \beta_d^{w1}(b_1\langle x_1\rangle \mid G) \cdots \beta_d^{w(i-1)}(b_{i-1}\langle x_{i-1}\rangle \mid G)$$
$$\cdot \beta_d^{w(i+1)}(b_{i+1}\langle x_{i+1}\rangle \mid G) \cdots \beta_d^{wk}(b_k\langle x_k\rangle \mid G)$$
$$\cdot p(e\langle h\rangle).$$

Presuming a phg-la G annotated with $H \cdot \{0,1\}$, and some derivation d annotated only with annotations from H, the inside and outside probabilities over d allow the following (compare Fig. 4.10): the inside probability $\beta_d^w(a\langle x\rangle \mid G)$ in d at some position w is the sum over the probabilities of all possible $H \cdot \{0,1\}$-annotations of the subtree $d|_w$ that have at their root a hyperedge with head $a\langle x\rangle$. Similarly, the outside probability $\alpha_d^w(a\langle x\rangle \mid G)$ sums up over the probabilities of all those possible annotations of the context enclosing the position w in d which have a tail node $a\langle x\rangle$ at the "gap" at their bottom.

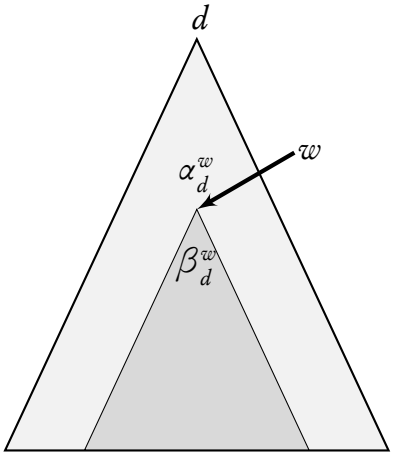


Figure 4.10: Inside and outside probabilities

With these definitions we have for every $d \in \partial(A_G)$, $w \in \mathrm{pos}(d)$, $a\langle x\rangle \in N\langle H\rangle$

$$\alpha_d^w(a\langle x\rangle \mid G) \cdot \beta_d^w(a\langle x\rangle \mid G) = P\big(\mathrm{hd}(X(w)) = a\langle x\rangle, Y = d \mid G\big), \tag{4.4}$$

presuming that $P(Y = d \mid G) > 0$, compare Manning and Schütze (1999, p. 399). This allows us to compute one of the probabilities from above as

$$\begin{aligned} &P(\mathrm{hd}(X(w)) = \mathrm{hd}(e\langle h\rangle) \mid Y = d, G) \\ &\quad = \frac{P(\mathrm{hd}(X(w)) = \mathrm{hd}(e\langle h\rangle), Y = d \mid G)}{P(Y = d \mid G)} && \text{(by Def. 2.8)} \\ &\quad = \frac{\alpha_d^w(\mathrm{hd}(e\langle h\rangle) \mid G) \cdot \beta_d^w(\mathrm{hd}(e\langle h\rangle) \mid G)}{P(Y = d \mid G)}. \end{aligned}$$

Note that this probability will necessarily be zero if $\mathrm{hd}(d(w)) \neq \mathrm{hd}(\partial(e\langle h\rangle))$, i.e., we can neglect summing over positions where this is the case, giving rise to a possible optimization.[10] If we expand the definition of β in (4.4), we can deduce, again under the assumption that $P(Y = d \mid G) > 0$, that

$$\begin{aligned} &\alpha_d^w(a\langle x_0\rangle \mid G) \cdot p(e\langle h\rangle) \cdot \beta_d^{w1}(b_1\langle x_1\rangle \mid G) \cdots \beta_d^{wk}(b_k\langle x_k\rangle \mid G) \\ &\quad = P\big(X(w) = e\langle h\rangle, Y = d \mid G\big) \end{aligned}$$

for every $d \in \partial(A_G)$, $w \in \mathrm{pos}(d)$, $e\langle h\rangle \in E\langle H\rangle$ with $\mu(e\langle h\rangle) = (b_1\langle x_1\rangle \cdots b_k\langle x_k\rangle, a\langle x_0\rangle)$, which, by similar reasoning, results in

$$\begin{aligned} &P(X(w) = e\langle h\rangle \mid Y = d, G) \\ &\quad = \frac{\alpha_d^w(a\langle x_0\rangle \mid G) \cdot p(e\langle h\rangle) \cdot \beta_d^{w1}(b_1\langle x_1\rangle \mid G) \cdots \beta_d^{wk}(b_k\langle x_k\rangle \mid G)}{P(Y = d \mid G)}, \end{aligned}$$

[10] As another optimization idea, one might compute the probability $P(Y = d \mid G)$ not in G, but in the less-split hypergraph G was produced from.

under the premise that $d(w) = \partial(e\langle h\rangle)$, else, $P(X(w) = e\langle h\rangle \mid Y = d, G) = 0$. Contrary to above, this requirement is necessary, because it allows to distinguish between two hyperedges e and e' with the same head and tail nodes: Without the premise, if e appears at position w in d, we could still give a different complete hyperedge $e'\langle h\rangle$ a non-zero likelihood of appearing in an annotated version of d. With these results, we can rewrite the fraction from (4.3) as

$$p_i(e\langle h\rangle) = \frac{\sum\limits_{\substack{d \in \partial(A_G^g) \\ P(Y=d|G_{i-1})>0}} f(d) \cdot \sum\limits_{\substack{w \in \mathrm{pos}(d) \\ d(w)=\partial(e\langle h\rangle)}} \alpha_d^w(a\langle x_0\rangle \mid G_{i-1}) \cdot p_{i-1}(e\langle h\rangle) \cdot \beta_d^{w1}(b_1\langle x_1\rangle \mid G_{i-1}) \cdots \beta_d^{wk}(b_k\langle x_k\rangle \mid G_{i-1}) / P(Y = d \mid G_{i-1})}{\sum\limits_{\substack{d \in \partial(A_G^g) \\ P(Y=d|G_{i-1})>0 \ \mathrm{hd}(d(w))=\mathrm{hd}(\partial(e\langle h\rangle))}} f(d) \cdot \sum\limits_{w \in \mathrm{pos}(d)} \alpha_d^w(a\langle x_0\rangle \mid G_{i-1}) \cdot \beta_d^w(a\langle x_0\rangle \mid G_{i-1}) / P(Y = d \mid G_{i-1})}.$$

The Inside-Outside algorithm then uses this reestimation formula instead of the more general one of EM, generating hypergraphs which capture more and more of the statistical properties of the supplied treebank, as shown in Alg. 4.3.

As stated earlier, this transformation demonstrates how Inside-Outside can be seen as a dynamic programming instance of EM: the computation of outside weights at a position wi of a derivation d depends on the outside weight at position w, as well as the inside weights of the sibling positions wk, $k \neq i$. Almost all of these values can be reused for the computation of the outside weights of a sibling position wj, $j \neq i$. In dynamic programming terminology, this property is referred to as *overlapping subproblems,* and calls for the *memoization* of their results. This simply means that in an implementation of Inside-Outside, one might want to manage a look-up table where the results of the functions α and β are stored indexed by their respective arguments, allowing efficient reuse in later computations.

For more information on memoization techniques in the context of Computational Linguistics, refer to Norvig (1991). Dynamic programming was first analyzed by Bellman (1954), with applications to several stochastic and probabilistic problems. Another comparison between EM and Inside-Outside was given by Prescher (2001).

Algorithm 4.3 The Inside-Outside algorithm for phg-las

Input: A phg-la $G = (N, E, H, \mu, g, p)$, a corpus $f : A_G^g \to \mathbb{R}$, $\partial : A_G \to \partial(A_G)$, random elements $X : \Omega \to A_G^g$ and $Y : \Omega \to \partial(A_G^g)$ and a value $\theta \in \mathbb{R}$.

Output: **Inside-Outside**$(G, f, \partial, X, Y, \theta)$

$i \leftarrow 0$

$G_0 = (N, E, H, \mu, g, p_0) \leftarrow G$

repeat

$i \leftarrow i + 1$

$G_i \leftarrow (N, E, H, \mu, g, p_i)$,

where for every $e\langle h\rangle \in E\langle H\rangle$ with $\mu(e) = (b_1\langle x_1\rangle \cdots b_k\langle x_k\rangle, a\langle x_0\rangle)$:

if $\sum_{\substack{d \in \partial(A_G^g) \\ P(Y=d \mid G_{i-1})>0}} f(d) \cdot \sum_{\substack{w \in \mathrm{pos}(d) \\ \mathrm{hd}(d(w))=\partial(a\langle x_0\rangle)}} \alpha_d^w(a\langle x_0\rangle \mid G_{i-1}) \cdot \beta_d^w(a\langle x_0\rangle \mid G_{i-1}) / P(Y = d \mid G_{i-1}) \neq 0$ **then**

$$p_i(e\langle h\rangle) = \frac{\sum_{\substack{d \in \partial(A_G^g) \\ P(Y=d \mid G_{i-1})>0}} f(d) \cdot \sum_{\substack{w \in \mathrm{pos}(d) \\ d(w)=\partial(e\langle h\rangle)}} \alpha_d^w(a\langle x_0\rangle \mid G_{i-1}) \cdot p_{i-1}(e\langle h\rangle) \cdot \beta_d^{w1}(b_1\langle x_1\rangle \mid G_{i-1}) \cdots \beta_d^{wk}(b_k\langle x_k\rangle \mid G_{i-1}) / P(Y = d \mid G_{i-1})}{\sum_{\substack{d \in \partial(A_G^g) \\ P(Y=d \mid G_{i-1})>0}} f(d) \cdot \sum_{\substack{w \in \mathrm{pos}(d) \\ \mathrm{hd}(d(w))=\partial(a\langle x_0\rangle)}} \alpha_d^w(a\langle x_0\rangle \mid G_{i-1}) \cdot \beta_d^w(a\langle x_0\rangle \mid G_{i-1}) / P(Y = d \mid G_{i-1})},$$

else

$p_i(e\langle h\rangle) = 1/|\{e'\langle h'\rangle \in E\langle H\rangle \mid \mathrm{hd}(e'\langle h'\rangle) = \mathrm{hd}(e\langle h\rangle)\}|$

end if

until $L\left(f; P(\cdot \mid G_i)\right) - L\left(f; P(\cdot \mid G_{i-1})\right) < \theta$

return G_i

5 Discussion: Induction of Heuristics

As described in Chapter 3, the KA* algorithm requires an external heuristic function, which is admissible and consistent, for its correct execution. One main aim of this work was to describe the method given by Pauls and Klein (2009b), in which they employ the hierarchy of probabilistic hypergraphs with latent annotations,[1] created during a run of the State-Split algorithm, to induce heuristics on the most-split hypergraph.

Let us first give an overview of the proposed method. Assume $G_1, \ldots, G_n$ are the probabilistic hypergraphs with latent annotations successively created during a run of State-Split. They are, w.l.o.g., presumed to be of the form $G_i = (N, E, H_i, \mu, g_i, p_i)$ for $i \in [n]$. The argumentation of Pauls and Klein is then based on the statement that G_i is a *relaxed projection* of G_{i+1}, for $i \in [n-1]$, which means that a split hyperedge's probability in G_{i+1} is never higher than the probability of the hyperedge in G_i it originated from:

$$p_i\big(e\langle x_1 \cdots x_k, x_0\rangle\big) \geq p_{i+1}\big(e\langle x_1 j_1 \cdots x_k j_k, x_0 j_0\rangle\big), \tag{5.1}$$

for every hyperedge $e \in E$, $j_0, \ldots, j_k \in \{0,1\}$, and $x_0, \ldots, x_k \in \{0,1\}^{i-1}$. With (5.1) and some straightforward induction proofs, this would then naturally lead to the fact that Viterbi outside scores of less-split nodes are higher than those of more-split ones emerging from them,

$$\alpha^*_{G_i}(a\langle x\rangle) \geq \alpha^*_{G_{i+1}}(a\langle x j\rangle), \tag{5.2}$$

for $i \in [n-1]$, $j \in \{0,1\}$, which would allow us to define a heuristic function h_i by

$$h_i : N\langle H_n\rangle \to [0,1]$$
$$h_i(a\langle x w\rangle) = \alpha^*_{G_i}(a\langle x\rangle).$$

for every $i \in [n-1]$, $x \in \{0,1\}^{i-1}$ and $w \in \{0,1\}^{n-i}$. It immediately follows from (5.2) that the h_i would be admissible, and the proof for their consistency would also arise to be quite trivial.

However, what did turn out to be not trivial at all, was the proof for (5.1), and this with good reason: at last, after a series of failed proof attempts, we decided to put it to the test using the in-house-developed *Vanda* project. Its implementation of State-Split was executed on a small fragment of the German language treebank NEGRA, with just one split-merge cycle. In the resulting two hypergraphs G_1 and G_2, there were already several hyperedges whose probabilities violate (5.1). Let us have a look at a minimal example distilled from this situation. Assume a treebank $f : T \to \mathbb{R}_{\geq 0}$ with only one type $T = \{t\}$,

$$t = \big\{a\big(b(c,c), b(d)\big)\big\},$$

[1] Actually, in their work, Pauls and Klein work directly on probabilistic context-free grammars, but since in the work at hand the definition for State-Split is based on hypergraphs, and pcfgs and phgs do not show differences regarding the following discussion, we will stick with the terminology introduced in Section 4.2.

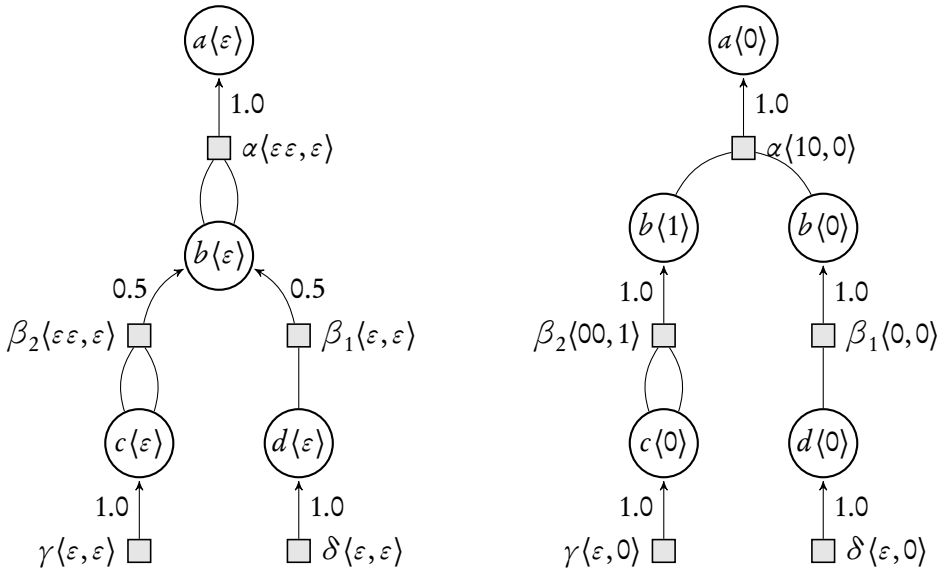


Figure 5.1: The phg-las G_1 and G_2 from the counterexample

and $f(t) = 1$.[2] The Vanda system reads off a probabilistic hypergraph with latent annotations[3] $G_1 = (N, E, H_1, \mu, a\langle\varepsilon\rangle, p_1)$ from this corpus, with nodes, hyperedges and annotation symbols

$$N = \{a, b, c, d\}, \quad E = \{\alpha, \beta_1, \beta_2, \gamma, \delta\}, \quad H_1 = \{\varepsilon\},$$

hyperedge connectivity given by

$$\begin{aligned} &\mu(\alpha) = (bb, a) \\ &\mu(\gamma) = (\varepsilon, c) && \mu(\delta) = (\varepsilon, d) \\ &\mu(\beta_1) = (d, b) && \mu(\beta_2) = (cc, b), \end{aligned}$$

and probabilities, arising from relative frequency estimation, as

$$\begin{aligned} &p_1(\alpha\langle\varepsilon\varepsilon, \varepsilon\rangle) = 1 \\ &p_1(\gamma\langle\varepsilon\rangle) = 1 && p_1(\delta\langle\varepsilon\rangle) = 1 \\ &p_1(\beta_1\langle\varepsilon, \varepsilon\rangle) = 0.5 && p_1(\beta_2\langle\varepsilon\varepsilon, \varepsilon\rangle) = 0.5. \end{aligned}$$

The phg-la G_1 is displayed in Fig. 5.1a. Running G_1 through a split-merge cycle of Vanda, the resulting phg-la is $G_2 = (N, E, H_2, \mu, a\langle 0\rangle, p_2)$ where $H_2 = \{0, 1\}$ and the new complete hyperedge probabilities are

$$\begin{aligned} &p_2(\alpha\langle 10, 0\rangle) = 1 \\ &p_2(\gamma\langle\varepsilon, 0\rangle) = 1 && p_2(\delta\langle\varepsilon, 0\rangle) = 1 \end{aligned}$$

[2] One might think of b as a constituent that can expand either to a composite phrase or a single utterance. Actually, this was the case in the NEGRA parse tree the example was distilled from, there b was labeled CVP (coordinated verb phrase) and could expand into "VP" or "VP and VP".

[3] Using the notation for phg-las established in the work at hand, instead of the one internally employed by Vanda

$$p_2(\beta_1\langle 0,0\rangle) = 1 \qquad\qquad p_2(\beta_2\langle 00,1\rangle) = 1,$$

and for all other complete hyperedges, their differences from zero are marginal. G_2 is shown in Fig. 5.1b, where nodes with negligible inside probabilities and edges with close-to-zero edge probabilities have been omitted. The juxtaposed hypergraphs are really quite instructive of the proceeding of the State-Split method: during the split-merge step, the two different "roles" the node $b\langle\varepsilon\rangle$ takes—once it dominates d, and once two nodes labeled with c in the treebank—were separated into distinct complete nodes $b\langle 0\rangle$ and $b\langle 1\rangle$, each now with only one ingoing (relevant) hyperedge, and each of those with a probability of one. Obviously, this split drastically increases the likelihood of f allocated by the hypergraphs' induced probability distributions (from 0.25 to 1.0), and hence it was not merged back together during the procedure. The other splits were merged back, however, and this makes sense, too, since they already perfectly grasped the single parse tree in the corpus's set of types.

By all means, G_2's edge probabilities are clearly in violation of (5.1), after all the hyperedges they were created from had both a probability of 0.5! This seriously impairs our hopes for the straightforward method described by Pauls and Klein to produce heuristics that are actually admissible and consistent. There are two possible explanations for the disagreement between (5.1) being stated as true by Pauls and Klein, and our counterexample.

First of all, one might think that the phenomenon does not carry weight for corpora larger than in this small example, with the cross-dependencies between trees ruling out such splits as for $b\langle 0\rangle$ and $b\langle 1\rangle$. But, at least at the level of lexical splits, the phenomenon should arise again, and in fact, this is witnessed by the experimental data written down by Petrov et al. (2006). There, phrasal categories are shown with the subcategories they were split into, along various semantic and syntactical characteristics. For example, the part-of-speech tag NNP (for singular proper nouns) exhibits sixteen subcategories after four split-merge cycles, with categories for names of months, of political leaders, and so on.

This strongly suggests that the probability of a hyperedge with, say, head "NNP-14"[4] and tail "December" is higher than of a hyperedge from "December" to "NNP" in the original hypergraph, after all there are more relevant ingoing hyperedges to NNP over which to "distribute" the total probability mass of one.

There might of course be another explanation for the discrepancy. Since the State-Split procedure was not formally defined by Petrov et al. in their paper, there is always the possibility that their concrete implementation differs from the formalization presented in the paper at hand. So, for an alternative version of State-Split, (5.1) might indeed hold. However, unfortunately, we were not able to extrapolate such an alternative version from the condensed presentation in their publication.

Therefore, the question of how to extrapolate consistent and admissible heuristics for KA* remains unanswered in the work at hand.

[4] In their paper, annotations are integers instead of words over $\{0,1\}$.

6 Summary

In Chapter 3 of the work at hand, we detailed the lazy version of the KA* algorithm for efficient heuristic-based k-best search in a probabilistic hypergraph. The basic monotony property this laziness relies on was illustrated, and the construction of ranked assignments with backpointers was given in the form of hyperedges of a search graph, similar to non-lazy KA*. After that, an implementation of KA* in the functional programming language Haskell was documented.

Chapter 4 concerned itself with grammars and hypergraphs: In Section 4.1, we gave a definition for probabilistic tree insertion grammars and showed how these can be transformed into probabilistic hypergraphs, such that derivations in the latter correspond to those of the grammar. Section 4.2 contained a description of the State-Split procedure for learning annotated phgs from a treebank, gave a short illustration of the Expectation-Maximization algorithm, and showed how the Inside-Outside algorithm can be seen as an instance of EM.

Finally, in Chapter 5, we considered a technique described by Pauls and Klein (2009b) to derive consistent and admissible heuristics for KA* from the hierarchy of hypergraphs generated by the State-Split procedure. Unfortunately, the property this technique rests on and which is stated as true by the authors, does not hold generally, at least for the case in the work at hand. We gave a counter-example for this property and concluded with a discussion of possible reasons for this, somewhat disappointing, result.

Bibliography

Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-45520-0.

James K. Baker. Trainable Grammars for Speech Recognition. *The Journal of the Acoustical Society of America*, 65(S1):S132, 1979. URL http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal&id=JASMAN0000650000S100S132000001&idtype=cvips&gifs=yes.

Richard Bellman. The Theory of Dynamic Programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, November 1954.

Eugene Charniak. Statistical Techniques for Natural Language Parsing. *AI Magazine*, 18: 33–44, 1997. URL http://aaaipress.org/ojs/index.php/aimagazine/article/viewFile/1320/1221.

John Chen and Vijay K. Shanker. *Automated Extraction of TAGs from the Penn Treebank*, pages 73–89. Kluwer Academic Publishers, Norwell, MA, USA, 2004. ISBN 1-4020-2293-X. URL http://dl.acm.org/citation.cfm?id=1139041.1139046.

Zhiyi Chi and Stuart Geman. Estimation of Probabilistic Context-Free Grammars. *Computational Linguistics*, 24(2):298–305, 1998.

Noam Chomsky. Three Models for the Description of Language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

Noam Chomsky. *Syntactic Structures*. Mouton, The Hague, 1957.

Alain Colmerauer and Philippe Roussel. The Birth of Prolog. *ACM SIGPLAN Notices*, 28:37–52, March 1993. ISSN 0362-1340. URL http://doi.acm.org/10.1145/155360.155362.

Arthur P. Dempster, Nan McKenzie Laird, and Donald B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, B(39):1–38, 1977.

Steve DeNeefe, Kevin Knight, and Heiko Vogler. A Decoder for Probabilistic Synchronous Tree Insertion Grammars. In *Proceedings of the 2010 Workshop on Applications of Tree Automata in Natural Language Processing*, ATANLP '10, pages 10–18, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics. ISBN 978-1-932432-79-4. URL http://portal.acm.org/citation.cfm?id=1870450.1870452.

Christy Doran, Dania Egedi, Beth Ann Hockey, B. Srinivas, and Martin Zaidel. XTAG System – A Wide Coverage Grammar for English, 1994.

Maya R. Gupta and Yihua Chen. Theory and Use of the EM Algorithm. *Foundations and Trends in Signal Processing*, 4:223–296, March 2011. ISSN 1932-8346. URL http://dx.doi.org/10.1561/2000000034.

Steffen Hölldobler. *Logik und Logikprogrammierung*. Kolleg Synchron, 2001. ISBN 9783935025591.

Aravind K. Joshi. Tree Adjoining Grammars: How Much Context-Sensitivity is Required to Provide Reasonable Structural Descriptions? In David R. Dowty, Lauri Karttunen, and Arnold Zwicky, editors, *Natural Language Parsing*, pages 206–250. Cambridge University Press, Cambridge, 1985.

Aravind K. Joshi and Yves Schabes. Tree-Adjoining Grammars and Lexicalized Grammars. Technical report, University of Pennsylvania, School of Engineering and Applied Science, Dept. of Computer and Information Science, 1991. URL http://books.google.com/books?id=LCTfGwAACAAJ.

Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree Adjunct Grammars. *Journal of Computer and System Sciences*, 10(1):136–163, 1975.

Olav Kallenberg. *Foundations of Modern Probability*. Probability and Its Applications. Springer, 2001. ISBN 9781441929495. URL http://books.google.com/books?id=cf2bcQAACAAJ.

Karim Lari and Steve J. Young. Applications of Stochastic Context-Free Grammars Using the Inside-Outside Algorithm. *Computer Speech & Language*, 5(3):237–257, 1991. ISSN 0885-2308. URL http://www.sciencedirect.com/science/article/pii/088523089190009F.

Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-13360-1.

Mitchell P. Marcus, Beatrice Santorini, and Mary A. Marcinkiewicz. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19, 1994. URL http://www.aclweb.org/anthology-new/J/J93/J93-2004.pdf.

Rebecca Nesson, Stuart Shieber, and Alexander Rush. Induction of Probabilistic Synchronous Tree-Insertion Grammars for Machine Translation. In *Proceedings of the 7th Conference of the Association for Machine Translation in the Americas (AMTA) 2006*, pages 128–137, 2006.

Peter Norvig. Techniques for Automatic Memoization with Applications to Context-Free Parsing. *Computational Linguistics*, 17:91–98, March 1991. ISSN 0891-2017. URL http://dl.acm.org/citation.cfm?id=971738.971743.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008. ISBN 9780596514983. URL http://book.realworldhaskell.org/read/.

Adam Pauls and Dan Klein. *k*-best A* Parsing. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2 - Volume 2*, ACL '09, pages 958–966, Stroudsburg, PA, USA, 2009a. Association for Computational Linguistics. ISBN 978-1-932432-46-6. URL http://portal.acm.org/citation.cfm?id=1690219.1690281.

Adam Pauls and Dan Klein. Hierarchical Search for Parsing. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 557–565, Boulder, Colorado, June 2009b. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/N/N09/N09-1063.pdf.

Fernando Pereira and Yves Schabes. Inside-Outside Reestimation from Partially Bracketed Corpora. In *Proceedings of the 30th Annual Meeting of the ACL*, pages 128–135. Morristown, NJ: Association for Computational Linguistics, 1992.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. Learning Accurate, Compact, and Interpretable Tree Annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 433–440, Sydney, Australia, July 2006. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P/P06/P06-1055.

Simon Peyton-Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003. ISBN 0521826144. URL http://www.worldcat.org/isbn/0521826144.

Detlef Prescher. Inside-Outside Estimation Meets Dynamic EM. Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, 2001. URL http://books.google.com/books?id=O1xPPwAACAAJ.

Detlef Prescher. A Tutorial on the Expectation-Maximization Algorithm Including Maximum-Likelihood Estimation and EM Training of Probabilistic Context-Free Grammars, March 2005. URL http://arxiv.org/abs/cs/0412015.

Yves Schabes. Stochastic Tree-Adjoining Grammars. In *Proceedings of the workshop on Speech and Natural Language*, HLT '91, pages 140–145, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics. ISBN 1-55860-272-0. URL http://dx.doi.org/10.3115/1075527.1075558.

Yves Schabes and Richard C. Waters. Tree Insertion Grammar: A Cubic-Time Parsable Formalism that Lexicalizes Context-Free Grammar without Changing the Trees Produced. *Computational Linguistics*, 21, 1994.

Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and Implementation of Deductive Parsing. *Journal of Logic Programming*, 24(1&2):3–36, 1995. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.8763.