PhD Depth Examination Report
Algebraic Foundation of Statistical Parsing

Semiring Parsing

Yudong Liu

200101828

Supervised by: Dr. Anoop Sarkar

Computing Science Department

Simon Fraser University

November 26, 2004

Abstract

Statistical parsing algorithms are useful in structure predictions, ranging from NLP to biological sequence analysis. Currently, there are a variety of efficient parsing algorithms available for different grammar formalisms. Conventionally, different parsing descriptions are needed for different tasks; a fair amount of work is required to construct for each one. Semiring parsing is proposed to provide a generalized and modularized framework to unify all these different parsing algorithms into a general framework and by separation of the algebra and the algorithms, it makes the very same algorithm can perform across diverse tasks.

One main concern about the semiring parsing system is the efficiency considerations. A packed representation for all possible target structures was discussed and different heuristic search strategies have been explored. By investigating more structured probabilistic models, we found that all the models are using the similar packed structures and apply the similar dynamic programming as classic inside-outside algorithms to the parameter estimation, which indicates that semiring parsing can be further extended to more complex models and can integrate more tasks, such as probabilistic learning.

Semiring parsing turns out to have a solid theoretical foundation and has a promising perspective of applications.

Contents

1 Introduction

Statistical parsing algorithms are useful for structure predictions in many diverse application areas, ranging from natural language processing to biological sequence analysis.

In natural language processing, typically, sentences are far more ambiguous than one might have thought. There may be hundreds, even millions, of syntactic parse trees for some natural sentences. Some methods have been proposed for dealing with such syntactic ambiguity in ways that exploit certain *regularities* among alternative parse trees. A special case is that ambiguity coefficients follow a well-known combinatorial series called the *Catalan Numbers*. Theoretically, such encoding of ambiguity indicates that parsing can be brought into an *algebraic power series* framework and there exists a solid algebraic foundation to define parsing as the algebraic operations in this framework. From the application point of view, parsing algorithms can be used to compute many interesting quantities, such as Viterbi value (the value of the best derivation), Viterbi derivation (the parse tree corresponding to the Viterbi value), n-best derivations, derivation forests, derivation count, inside and outside probabilities of components of the input string. In particular, currently there are a variety of efficient parsing algorithms available for different grammar formalisms, from context-free grammar to tree adjoining grammar. Conventionally, different parsing descriptions are needed for different tasks; a fair amount of work can be required to construct each one. Therefore it is preferable to unify all these parsers into a general framework to make it work across diverse tasks and application areas.

Semiring parsing is motivated and proposed by the discussions above. It has a solid algebraic foundation and its soundness and completeness have been proved both in theory and in practice. This framework is based on the theory of power series, and takes statistical parsing to be the computation of the *coefficients* for the elements of a free monoid. The monoid typically represents a set of languages. Roughly speaking, a semiring parsing system consists of a deductive component and a semiring interpreter component. A deductive system provides us a unified way to represent a variety of parsing algorithms and particularly suitable for rapid prototyping new parsing strategies. And a semiring interpreter component assigns the corresponding semantics to the results of deductive system by defined semiring specializations.

Therefore, by separation of the algebra and the parsing algorithms, a semiring parsing system provides a generalized and modularized framework to unify a variety of parsing algorithms across a variety of tasks.

One main concern about semiring parsing is the issue of efficiency. In general the number of the parses may be exponential in the size of the input. To make the semiring parsing system more practical, we have to find a packed representation of all possible parses without enumerating them one by one. And to minimize the observed parse time, we hope to locate the target parse faster and more accurately by using a good search strategy while parsing. All these issues will be explored in the report.

By investigating many formalisms of structured probabilistic models in statistical parsing, we found that all the formalisms currently studied are using analogous packed representations and applying similar dynamic programming algorithms on the packed structures. Furthermore, all these formalisms perform parameter estimation by using some variants of the classic Inside-Outside algorithm on the packed structures. And these formalisms enable more powerful and general statistical models, such as Markov Random Fields, to do parsing and probabilistic grammar learning. All this provides a wider perspective for semiring parsing and also provides a possibility to further generalize semiring parsing. Based on these more general formalisms, to formalize more tasks, such as probabilistic grammar learning, into the semiring framework is one of our future work.

A preliminary semiring parsing system has been implemented. A more efficient implementation and more novel applications in different domains may be created based on the framework as the future work.

The report is organized as follows: After reviewing some basics of formal language theory, regular languages and context-free languages (Section 2, Section 3 and Section 4 respectively), an algebraic foundation for formal languages is discussed in detail. In particular, the properties of two particular types of power series corresponding to regular languages and context-free languages respectively are explored from the literature (Section 5). Section 6 gives the concepts for probabilistic context-free grammars (PCFGs) and the classic parsing algorithms for PCFGs. In Section 7, we introduce the components of deductive parsing and give some examples to illustrate the working system. Based on the previous discussions, Section 8 gives the principles of semiring parsing systems and the implementation details for a working preliminary semiring parsing system. In Section 9, we mainly discuss the current work of exploring efficient parsing. In Section 10, more structured probabilistic models are presented to give a wider and further perspective beyond our current formalism (PCFGs) and parsing representation (dependency graph). Section 11 will summarize the report and talk about some of future directions. Appendix A gives more formal notations for algorithms for PCFGs.

2 Elements of Formal Language Theory

This section gives a general introduction to the notation used in this depth report and some basic concepts we use from formal language theory.

2.1 Symbols, Alphabets, Strings and Languages

- An *alphabet* Σ is a finite nonempty set. The elements of Σ are called *symbols*. Letter and digits are examples of frequently used symbols. e.g., $\Sigma = \{a, b, c\}$.

- A *string* (or *word*) over alphabet Σ is a finite length string of symbols taken from Σ. For example, $\Sigma = \{a, b, c\}$ and $abcb$ is a string. The *length* of a string w, denoted $|w|$, is the number of symbols composing the string. e.g., $|abcb| = 4$. The string consisting of zero symbols is called *empty string*, written ϵ. Thus $|\epsilon| = 0$.

- The set of all strings (resp. all nonempty strings) over the alphabet Σ is denoted by Σ^* (resp. Σ^+).

- A *prefix* of a string is any number of leading symbols of that string, and a *suffix* is any number of tailing symbols. e.g., string abc has prefix ϵ, a, ab, and abc; its suffixes are ϵ, c, bc, and abc. A prefix (resp. suffix) of a string, other than the string itself, is called a *proper* prefix (resp. suffix).

- For strings w_1 and w_2, the juxtaposition w_1w_2 is called *concatenation*. Concatenation is associative and has an identity ϵ. That is,

Associativity	$(uv)w = u(vw)$
Unit	$\epsilon w = w\epsilon = w$ for each string w

- Subsets of Σ^* are referred to as *(formal) languages* over the alphabet Σ.

Various unary and binary *operations* defined for languages will be considered in the sequel. Regarding languages as sets, we use the standard set operations of union, intersection, complementation, difference and symmetric difference[1].

[1] In mathematics, the symmetric difference of two sets A and B is the set of elements which are in one of either set, but not in both. The most common notation used for symmetric difference is $A \triangle B$.

- The *concatenation* (or *product*) of two languages L_1 and L_2, in symbols L_1L_2 is defined by

$$L_1L_2 = \{w_1w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}.$$

- The notation L^i is extended to concern the concatenation of languages. LL is written as L^2. This notation is generalized to

$$L^0 = \{\epsilon\},\ L^1 = L \text{ and } L^i = LL^{i-1} = L^{i-1}L \text{ for } i \geq 2$$

the *Kleene closure* of L (resp. *ϵ-free concatenation closure*), written L^* (resp. L^+) , is then defined by

$$L^* = \bigcup_{i=0}^{\infty} L^i \quad (\text{resp.} \quad L^+ = \bigcup_{i=1}^{\infty} L^i)$$

and hence $L^* = L^+ \cup \{\epsilon\}$.

- We now define the operation of *substitution*. For each symbol a of an alphabet Σ, let $\sigma(a)$ be a language. Define,

$$\sigma(\epsilon) = \{\epsilon\}, \quad \sigma(w_1w_2) = \sigma(w_1)\sigma(w_2), \text{ for } w_1, w_2 \in \Sigma^*.$$

For a language L over Σ, we define

$$\sigma(L) = \{u \mid u \in \sigma(w) \text{ for some } w \in L\}.$$

Such a mapping σ is called a *substitution*. A substitution σ is *ϵ-free* iff none of the languages $\sigma(a)$ contains the empty string. A substitution σ such that each $\sigma(a)$ consists of a single string is called a *homomorphism*. A symbol-to-symbol homomorphism will often be called a *coding*.

2.2 Language Representations – Generators and Recognizers

The main objects of language representations are finitary specifications of infinite languages. Most of such specifications are obtained as special cases from the notion of a *rewriting system*. By definition, a *rewriting system* is an ordered pair (Σ, P), where Σ is an alphabet and P a finite set of ordered pairs of strings over Σ. The elements (w, u) of P are referred to as *rewriting rules* or *productions* and

denoted by $w \rightarrow u$. Given a rewriting system, the *derivation relation (or yield relation)* $\Rightarrow$ in the set Σ^* is defined as follows.

For any strings w and u, $w \Rightarrow u$ holds iff there are strings w', w_1, w_2, u_1 such that $w = w'w_1w_2, u = w'u_1w_2$, and $w_1 \rightarrow u_1$ is a production in the system. The *reflexive transitive closure* (resp. transitive closure) of the relation $\Rightarrow$ is denoted by $\Rightarrow^*$ (resp. $\Rightarrow^+$).

In this part, we shall discuss from a general point of view the two principal methods of defining languages– the generator and the recognizer, which typically correspond to the grammar and the automata respectively.

2.2.1 Language Generators: Formal grammars

Grammars are the most important class of generators of languages. It defines a language in a mathematical way and gives the sentences in the language useful structures. In the following, we mainly focus on the phrase structure grammars (called Chomsky grammars sometimes).

Phrase Structure Grammars A *phrase structure grammar* is a 4-tuple $G = (N, \Sigma, P, S)$, where

- N is a finite set of *nonterminal symbols*;
- Σ is a finite set of *terminal symbols* that is disjoint from N;
- Additionally, set $V = \Sigma \cup N$, called the *vocabulary*;
- Again, the elements of P are referred to as rewriting rules or productions and of the form
$$(\Sigma \cup N)^+ \rightarrow (\Sigma \cup N)^*$$
- Symbol S in N is indicated as the *start symbol*.

A phrase structure grammar G as above defines a rewriting system (V, P). Let $\Rightarrow$ and $\Rightarrow^*$ be the relations determined by this rewriting system. Then the language $L(G)$ *generated* by G is defined by :

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}.$$

Two grammars G and G' are termed *equivalent* iff $L(G) = L(G')$.

A special kind of string called a *sentential form* of a grammar G is defined as: if α is a string over V (recall $V = \Sigma \cup N$), and $S \Rightarrow^* \alpha$, then α is a *sentential form*. Particularly a sentential form containing no nonterminal symbols is a *sentence* generated by G.

Example 2.1 Let G be defined by:

$$\begin{aligned} S &\rightarrow XX \\ X &\rightarrow AA \\ X &\rightarrow A \\ A &\rightarrow a \end{aligned}$$

We have the following derivations in G:

$$\begin{aligned} S &\Longrightarrow XX \\ &\Longrightarrow AAX \\ &\Longrightarrow aAX \\ &\Longrightarrow aaX \\ &\Longrightarrow aaA \\ &\Longrightarrow aaa \end{aligned}$$

The language generated by G is: $G(L) = \{a^n | n \geq 1\}$.

Some sentential forms derived from G are S, XX, AAX, aAX

Restricted Grammars In terms of different productions forms, grammars can be classified into four levels numbered from 0 to 3. A grammar $G = (N, \Sigma, S, P), V = \Sigma \cup N$ is of the type i iff the restrictions i on P, as given below, are satisfied:

1. No restrictions.
2. Each production in P is of the form $\alpha \rightarrow \gamma$, where $|\gamma| \geq |\alpha|$, and $\alpha, \gamma \in V^*, \alpha \neq \epsilon$. An alternative presentation is: $\alpha A \beta \rightarrow \alpha \gamma \beta$, where $\alpha, \beta, \gamma \in V^*, \gamma \neq \epsilon$ and $A \in N$ (with the possible exception of the productions $S \rightarrow \epsilon$ whose occurrence in P implies, however, that S does not occur on the right side of any production).
3. Each production in P is of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in V^*$.

4. Each production is of one of the two forms $A \rightarrow aB$ or $A \rightarrow a$, where $A, B \in N$ and $a \in \Sigma^*$.

A language is of type i iff it is generated by a grammar of type i. Type 0 grammars and languages are also called *recursively enumerable*. Type 1 grammars and languages are also called *context-sensitive*. Type 2 grammars and languages are also called *context-free*. Type 3 grammars and languages are also referred to as *right-linear* or *regular*.

These four types of grammars are all proper inclusion, which means every regular language is context-free, every context-free language is context-sensitive, every context-sensitive language is recursive and every recursive language is recursively enumerable. More discussion about this classification can be found in most literatures of formal language theory. In this report, we shall be mainly interested in the latter two classes.

2.2.2 Language Recognizers: Automata

Grammars are a *generative* formalism, i.e., it is easy to build a string in the language described by a grammar, but it is less easy to *decide whether a given string is in the language described by a grammar* which is often called *a recognition problem*.

Language recognizers are a second common method of providing a finite specification for a language. A recognizer defines a language L by receiving arbitrary word as input and accepts exactly those words belonging to L. The four types of grammars defined above can be obtained also by recognizers. Typically, recognizers are *finite state automata* and *push-down automata* corresponding to regular languages and context-free languages; Context-sensitive languages and recursive enumerable languages corresponds to *linear-bounded automata* and *Turing machines*. In the following, we just give a descriptive definition of the general structure of recognizers. More formal definitions of finite state automata and push-down automata will be introduced later.

In general, there are three components for a recognizer – an input tape, a finite state control, and an auxiliary memory. It can be pictured as follows:

Input Tape The *input tape* can be divided into a linear sequence of tape squares, each tape square containing exactly one input symbol from a finite input alphabet. There is an *input head*, which can read one input square at a given instant time. In

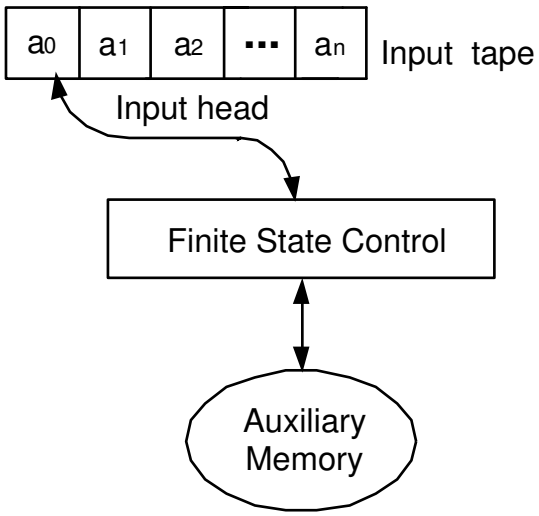


Figure 2.1: *A recognizer.*

a move by a recognizer, the input head can move one square to the left, or to the right, or remain stationary.

Auxiliary Memory The *memory* of a recognizer can be any type of data structure. Generally, we assume that there is a finite *memory alphabet* and the memory contains only the symbols from that alphabet. An important example of auxiliary memory is the pushdown stack, which can be represented as a string of memory symbols, e.g., $Z_1 Z_2 \cdots Z_n$ where each Z_i is assumed to be from the finite memory alphabet Γ and Z_1 is on the top of the stack.

The behavior of the auxiliary memory for a class of recognizers is characterized by two functions – a store function and a fetch function.

The *fetch function* is a mapping from the set of memory configurations to a finite set of information symbols. For example, the only information that can be accessed from a pushdown stack is the topmost symbol. Thus a fetch function f for a pushdown stack would be a mapping from Γ^+ to Γ such that $f(Z_1 Z_2 \cdots Z_n) = Z_1$.

The *store function* is a mapping which describes how memory may be altered. It maps memory and a *control string* to memory. For example, if we assume that a store operation for a pushdown stack replaces the topmost symbol on the pushdown stack by a finite length string of memory symbols, then the store function g

could be represented as $g : \Gamma^+ \times \Gamma^* \rightarrow \Gamma^*$, such that

$$g(Z_1 Z_2 \cdots Z_n, Y_1 \cdots Y_k) = Y_1 \cdots Y_k Z_2 \cdots Z_n.$$

If we replace the topmost symbol Z_1 on the pushdown stack by the empty string, then the symbol Z_2 becomes the topmost symbol and can then be accessed by the fetch operation.

Finite State Control The heart of a recognizer is the *finite state control*, which can be thought of as a procedure which controls the behavior of the recognizer. The control can be represented as a finite set of states together with a mapping which describes how the states changes according to the current input symbol and the current information the fetch function returns. The control also determines in which direction the input head moves along the input tape and what information is going to be stored into the memory. A recognizer operates by making a sequence of moves, which is basically the steps of the procedure execution. Conventionally, we use *configurations* to describe the behavior of a recognizer. A configuration is a picture of the recognizer describing:

1. the state of the finite control;
2. the contents of the input tapes, the location of the input head;
3. the contents of memory.

The *initial configuration* of a recognizer is one in which the finite control is in a specified initial state, the input head is scanning the leftmost symbol of the input tape, and the memory has a specified initial content. A *final configuration* is one in which the finite control is in one of a specified set of final states and the input head is at the end of the tape (if any). Often, the memory must also satisfy certain conditions if the configuration is to be considered final.

We say that a recognizer *accepts an input string* w if, starting from the initial configuration with w on the input tape, the recognizer can make a sequence of moves and end in a final configuration.

Note that the finite state control of a recognizer can be deterministic or nondeterministic. The control is *deterministic* if for each configuration there is at most one possible move. A nondeterministic recognizer may be able to make many different sequences of moves from an initial configuration. However, if at least one of these sequences ends in a final configuration, then the initial input string

will be accepted.

The language defined by a recognizer is the set of input strings it accepts.

2.3 The Chomsky Hierarchy

Till now, we have given a general description of two representations of formal languages – grammars as generators and automata as recognizers. The classification of formal grammars is typically referred to the Chomsky hierarchy (Noam Chomsky 1956). And for each of the four grammars in the hierarchy there is a natural classes of recognizers that define the same classes of languages. These recognizers are *Turing machines, linear bounded automata, pushdown automata, finite automata* in accordance with type 0 to 3 grammars. The following table gives a more distinct picture of Chomsky hierarchy.

Grammar	Language	Automata	Production Rule	Recognition Complexity Class
Type 0	Recursive enumerable languages	Turing machine	No restriction	Undecidable
Type 1	Context-sensitive languages	Linear-Bounded automata	$\alpha A\beta \rightarrow \alpha\gamma\beta$	NP-Complete
Type 2	Context-free languages	Push-down automata	$A \rightarrow \alpha$	Polynomial
Type 3	Regular languages	Finite automata	$A \rightarrow a\gamma, A \rightarrow a$	Linear

Table 1 Chomsky Hierarchy

From Table 1, we can see that recognition algorithms for regular languages and context-free languages belong to tractable complexity classes. For these reason, these two classes have become the most extensively studied formalisms. We should also mention that there are some other grammar formalisms that have been well-established in between type 1 and type 2 grammars in the Chomsky hierarchy, such as the notion of "mildly context-sensitive grammars", which refers to

a set of *weakly equivalent*[2] formalisms, namely, tree-adjoining grammar (TAG), linear indexed grammars (LIG) and combinatory categorial grammars (CCG). Mildly context-sensitive grammars are more powerful in describing natural languages compared with context-free grammars while remaining efficiently parsable in the general case. Another formalism – range concatenation grammar (RCG) has also been studied to reveal many attractive properties in natural language processing. They are powerful since they strictly contain the class of mildly context-sensitive formalisms and exactly covers the class *PTIME* of languages recognizable in deterministic polynomial time (Barthelemy et al., 2001b), which is called *PTIME complete*.

In this report, we are going to focus on the context-free languages and some of their algebraic properties. For the convenience of later reference, we first review some basic facts concerning regular languages and finite state automata.

3 Regular Languages (RLs), Finite Automata

In this section, we review some basic facts in regular languages and finite automata, mainly in order to fix notations and to allow reference in later sections. For most proofs omitted, they can be found in (John E. Hopcroft, 1979).

Regular languages are a class of languages central to much of language theory. There are three equivalent ways of specifying regular languages, namely *regular expressions, regular grammars* and *finite state automata*. These are "equivalent" in the sense that each defines precisely the same set of regular languages. Moreover there exist algorithms for transforming these three forms from one to the another.

3.1 Regular Grammars, Regular Expressions and Finite State Automata

In this part, we just briefly overview the concepts of regular grammars, regular expressions and finite state automata by giving a simple example. More detailed and formal introductions can be found in any literature book of formal language theory.

[2]Two formalisms are *weakly equivalent* if they correspond to the same set of languages.

Example 3.1 Let regular grammar $G = (\Sigma, N, P, S)$, where $\Sigma = \{a, b\}$, $N = \{S, A, B\}$ and P is

$$\begin{aligned} S &\rightarrow aA \mid bB \\ A &\rightarrow aA \mid b \\ B &\rightarrow bB \mid a \end{aligned}$$

The regular language $L(G) = \{ab, ba, aab, bba, aaab, bbba, \cdots\}$ which contains the strings characterized with $a's$ as prefix followed by a single b as suffix or $b's$ as prefix followed by a single a as suffix.

The regular language $L(G)$ can be equivalently represented by the regular expression $aa^*b) + bb^*a$. Readers can justify by themselves that the regular expression and regular grammar G represent exactly the same language.

Alternatively, Figure 3.1 shows the finite state automaton which recognizes exactly the language $L(G)$.

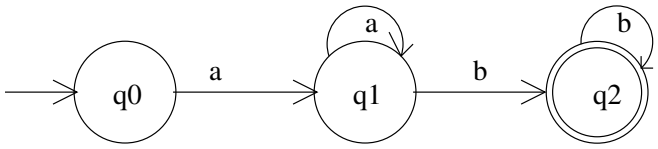


Figure 3.1: *finite state automata.*

Figure 3.1 shows an example of *deterministic* finite state automata (DFSA), in which every transition on the original state and the input symbol is unique. Accordingly a *nondeterministic* finite automaton (NFSA) is one that the new state obtained after reading an input symbol may not be uniquely determined. It has been proved that any NFSA can be transformed into an equivalent DFSA w.r.t they accept the same set of languages.

In summary, the following statements are equivalent:
L is a regular language
iff L can be generated by a regular grammar.
iff L can be described by a regular expression.
iff L can be accepted by a DFSA.
iff L can be accepted by an NFSA.

3.2 Finite State Transducers

Finite state transducers are basically finite state automata in which each transition has an output as well as the more familiar input.

Definition A *finite transducer* M is a 6-tuple $(Q, \Sigma_1, \Sigma_2, \delta, q_0, F)$ where

1. Q is a finite set of *states*;
2. $q_0 \in Q$ is the *initial state*;
3. $F \subseteq Q$ is the set of *final states*;
4. Σ_1 is a finite *input alphabet*;
5. Σ_2 is a finite *output alphabet*;
6. δ is a mapping $\delta : Q \times \Sigma_1 \times Q \rightarrow \Sigma_2$.

Figure 3.2 shows an example of a finite state transducer, the input alphabet is $\{a, b\}$, the output alphabet is $\{0, 1\}$, the transition functions are:

$$\begin{aligned} \delta(q_0, a, q_1) &= 1 \\ \delta(q_1, a, q_1) &= 0 \\ \delta(q_1, b, q_2) &= 0 \\ \delta(q_2, b, q_2) &= 1 \end{aligned}$$

when the input string is abb, we can get the output 101.

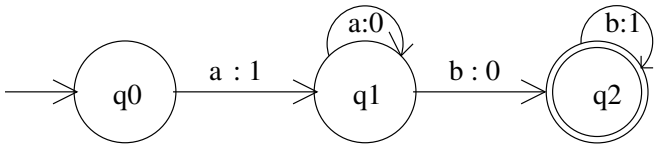


Figure 3.2: *A finite state transducer.*

4 Basics of Context-free Languages (CFLs)

In this section, we introduce basics of context-free grammars and pushdown automata.

4.1 Context-free Grammars (CFGs)

Context-free grammars (CFGs) $G = (\Sigma, N, P, S)$ is the most extensively studied formal grammar; it is powerful enough to describe the syntax of programming languages, and it is simple enough to allow the construction of efficient recognition algorithms. Recall that its production rules are of form $A \rightarrow \alpha$ where A is a non-terminal symbol and α is a sentential form consisting of terminals and/or non-terminals. The term "context-free" comes from the feature that the variable A can always be replaced by α, in no matter what context it occurs. A formal language is *context-free* if there is a context-free grammar which generates it.

4.1.1 Normal Forms of CFGs

Chomsky normal form (CNF) and *Greibach normal form* (GNF) are two normal forms of context-free grammars. Every context-free language which is ϵ-free (i.e., does not contain the empty string) can be generated by a grammar in CNF and GNF. A context-free grammar in Chomsky normal form (CNF) have all production rules of the form: $A \rightarrow BC$ or $A \rightarrow a$ where A, B and C are nonterminal symbols and a is a terminal symbol.

A context-free grammar in Greibach normal form (GNF) have all production rules of the form: $A \rightarrow a\alpha$ where A is a nonterminal symbol, a is a terminal symbol and α is (possibly empty) sequence of nonterminal symbols.

Every grammar in CNF/GNF is context-free, and conversely, every ϵ-free context-free grammar can be transformed into an equivalent one in CNF/GNF. "Equivalent" here means that the two grammars generate the same language.

Because of special form of production rules in CNF, this normal form has both theoretical and practical implications. For instance, given a context-free grammar in CNF, one can use it to construct a polynomial-time recognition algorithm. We will see such a context-free recognition algorithm called CKY later on.

GNF is useful for proving the equivalence of context-free language and nondeterministic pushdown automaton, and is also applied to prove Shamir's Theorem and the Chomsky-Schützenberger Theorem (G. Rozenberg, 1997).

4.1.2 Derivation Trees, Ambiguity of Context-free Languages

We have given the definition of *derivations* in Section 2.2. Derivations in context-free grammars are often visualized by *derivation trees*.

Example 4.1 Reuse the grammar G in example 1.1, the possible three derivations for string aaa are:

$$S \Rightarrow XX \Rightarrow AAX \Rightarrow aAX \Rightarrow aaX \Rightarrow aaA \Rightarrow aaa$$

$$S \Rightarrow XX \Rightarrow XAA \Rightarrow XAa \Rightarrow Xaa \Rightarrow Aaa \Rightarrow aaa$$

$$S \Rightarrow XX \Rightarrow AX \Rightarrow aX \Rightarrow aAA \Rightarrow aaA \Rightarrow aaa$$

The derivation trees that correspond to these three derivations are (Figure 4.1):

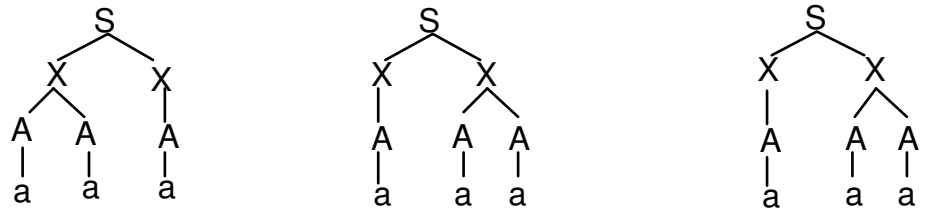


Figure 4.1: *Derivation trees.*

Observations:

1. The distinction between the first two derivations is that derivation one always picks the leftmost nonterminal to replace, but derivation two always picks the rightmost. So according to choosing which nonterminal to replace, we call the first kind of derivation *leftmost* derivation and the second *rightmost* derivation. In general, we say there are no essential distinct between these two kinds of derivations. We use leftmost derivations by default without loss of generality.

2. If we consider only leftmost derivations, then the first and the third derivation tree both are possible parse trees for string aaa. In this case, we say grammar G is *ambiguous*. Formally, a grammar is *ambiguous* if there is

more than one parse tree for some sentence. And a language L is *inherently ambiguous* if it can be generated only by ambiguous grammars. Disambiguation for ambiguous grammars is a basic problem in computational linguistics.

4.2 Pushdown Automata

Pushdown automata (PDA) is a recognition device which is equivalent to context-free grammars. For every CFG there is an equivalent PDA and visa versa. PDA's are important in the study of recognition algorithms for context-free languages.

4.2.1 Description, Definition and Example

Compared with FSAs, pushdown automata consist of an input tape, a finite control and a pushdown stack. The *pushdown stack*, just as the "stack" in usual sense, is a "first in-last out" linear list with two primitive operations that can be performed on the top of a stack – PUSH and POP. For our purpose, the symbols on the stack will be from a particular alphabet and the stack will also have an unlimited memory capacity. This condition ensures that it will always be possible to perform a PUSH operation without getting an *overflow* situation. However, when the stack is empty, a POP operation will still result in a *underflow* failure.

Informally, a (nondeterminsitic) pushdown automata (NPDA) can be described as a nondeterministic finite state automaton equipped with a pushdown stack, in which the top element can influence the transition function.

Definition Formally, an NPDA is a 7-tuple, $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where

1. Q is a finite set of *states*;
2. Σ is a finite alphabet called the *input alphabet*;
3. Γ is a finite alphabet called the *pushdown alphabet*;
4. $q_0 \in Q$ is the *initial state*;
5. $Z_0 \in \Gamma$ is a particular pushdown symbol called the *start symbol* which is initially the only symbol on the stack;
6. $F \subseteq Q$ is the set of *final states*;

7. δ is mapping from $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$.

A configuration of an NPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, at any time can be described by an ordered pair in $Q \times \Gamma^*$. The first element in the ordered pair gives the current state of the automaton and the second element gives the current stack contents.

Example 4.2 We will construct an NPDA M_1, such that $L(M_1) = \{ww^R \mid w \in \{a, b\}^+\}$. It is a palindrome language. For each string in the language, the first half of the string is as the same as the second half in a reverse order. Since we process the input one symbol at one time and we do not know the total length of the input until it has been all processed, we cannot tell when to stop pushing the symbols and start matching (popping). For this reason, the automaton has to be nondeterministic.

We set $M_1 = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b, \perp\}, \delta, q_0, \{q_2\})$ where $\perp$ acts as a marker for the bottom of the stack. The pushdown function δ is defined as follows:

$$\begin{aligned}
\delta(q_0, a, a) &= \{(q_0, aa), (q_1, \epsilon)\} \\
\delta(q_0, a, b) &= \{(q_0, ab)\} \\
\delta(q_0, b, a) &= \{(q_0, ba)\} \\
\delta(q_0, b, b) &= \{(q_0, bb), (q_1, \epsilon)\} \\
\delta(q_0, a, \perp) &= \{(q_0, a\perp)\} \\
\delta(q_0, b, \perp) &= \{(q_0, b\perp)\} \\
\delta(q_1, a, a) &= \{(q_1, \epsilon)\} \\
\delta(q_1, b, b) &= \{(q_1, \epsilon)\} \\
\delta(q_1, \epsilon) &= \{(q_2, \epsilon)\}
\end{aligned}$$

Theorem (a) *If $L = L(M)$ for some NPDA M, then L is a context-free language.*
(b) *If L is a context-free language, then there exists an NPDA M, such that $L = L(M)$.*

4.2.2 Deterministic Pushdown Automata (DPDA)

If the pushdown automata can make at most one move in any configuration then it is called *deterministic*. Based on the definition of nondeterministic pushdown automata, we have more conditions for all $q \in Q, a \in \Sigma$ and $Z \in \Gamma$ such as:

1. $\delta(q, a, Z)$ contains at most one element;

2. $\delta(q, \epsilon, Z)$ contains at most one element;

3. if $\delta(q, \epsilon, Z) \neq \emptyset$ then $\delta(q, a, Z) = \emptyset$ for all $a \in \Sigma$.

A language accepted by a DPDA is called a *deterministic context-free language* (DCFL) or, more commonly, just a *deterministic language*. Unfortunately, not every CFL is deterministic but the DCFLs are nevertheless a very important class of languages. If a CFL can be accepted by a DPDA then the solution of the derivation problem is made considerably easier. The syntax definition of most modern programming languages are designed so that the programs in the language can be reasonably easily compiled.

As known, every nondeterministic finite state automata can be determinized and accepts the same set of regular languages. Unfortunately, the same conclusion does not hold for pushdown automata, which means DPDA is a proper subset of NPDA.

4.3 Comparisons of Closure Properties of RLs, DCFLs and CFLs

Theorem (a) *Every regular language is deterministic but there are deterministic languages which are not regular;*

(b) *Every deterministic language is context-free but there are context-free languages which are not deterministic.*

The following table summarizes some of the known closure properties of regular languages, deterministic languages and context-free languages.

Closed under	Regular	Deterministic	Context-free
union	yes	no	yes
concatenation	yes	no	yes
Kleene closure	yes	no	yes
intersection	yes	no	no
intersection with regular languages	yes	yes	yes
complement	yes	yes	no
homomorphism	yes	no	yes

Table 2 Closure properties of RLs, DCFLs and CFLs

5 Weighted Automata and Their Algebraic Foundations

Weighted automata are more general devices than their unweighted counterparts in that their transitions are assigned with weights in addition to the usual alphabet symbols. Informally, a weighted automaton is a mapping from strings over an alphabet to weights. For example, when weights represent probabilities and assuming appropriate normalization, a weighted language is just a probability distribution over strings. A usual (unweighed) automaton is a particular kind of weighted automata with the weights being 0 or 1 only.

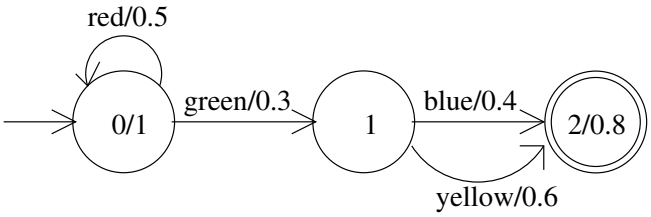


Figure 5.1: *Weighted finite automata.*

Let us look at an example of weighted finite state automata (WFSA). In Figure 5.1 each transition is attached with an input and a weight. And the final state is labeled with a weight as well. Here the weight on each transition denotes the probability of each transition with the given input.

5.1 Semirings and Weighted Automata

To make the operations on weighted automata well-defined, the weight set needs to have the algebraic structure of a *semiring*.

Definition A *monoid* consists of a set Σ, an associative binary operation $*$ on Σ and of an identity element $\bar{1}$ such that $\bar{1} * a = a * \bar{1} = a$ for every $a \in \Sigma$. A monoid is *commutative* if for every $a, b \in \Sigma, a * b = b * a$.

Definition A *semiring* $(K, \oplus, \otimes, \bar{0}, \bar{1})$ is defined such that

(i) $(K, \oplus, \bar{0})$ is a commutative monoid with $\bar{0}$ as the identity element for $\oplus$,

(ii) $(K, \otimes, \bar{1})$ is a monoid with $\bar{1}$ as the identity element for $\otimes$,

(iii) $\otimes$ distributes over $\oplus$: for all $a, b, c \in A$,

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

$$c \otimes (a \oplus b) = (c \otimes a) \oplus (c \otimes b)$$

(iv) $\bar{0}$ is an annihilator for $\otimes$: $a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$ for every a in A.

A semiring is said to be *commutative* when the multiplicative operation $\otimes$ is commutative: $\forall a, b \in A, a \otimes b = b \otimes a$. One of the most important semirings is *Boolean semiring* $(\{0, 1\}, \vee, \wedge, 0, 1)$. Another important semiring $(R_+ \cup \{\infty\}, \min, +, \infty, 0)$ is called *tropical semiring*. It is widely used in solving the "shortest-path problem" due to the fact that it fits the underlying algebraic structure of many classical shortest-paths algorithms. We will discuss these semirings in detail in Section 8. Obviously, both Boolean semiring and tropical semiring are commutative.

Definition A *weighted automaton* over the semiring K is a 7-tuple $A = (\Sigma, Q, I, F, \delta, \lambda, \rho)$ where Σ is the finite alphabet of the automaton, Q is a finite set of states, $I \subseteq Q$ is the set of initial state, $F \subseteq Q$ the set of final states, $\delta : Q \times \Sigma \times K \to Q$ a set of transition functions, $\lambda : I \to K$ the initial weight function mapping I to K, and $\rho : F \to K$ the final weight function mapping F to K.

We can generalize the definition of semirings: the weights in weighted automata do not need to be numbers, but they may instead be strings, sets, or even regular expressions. In each case, the semiring will be specialized differently. For example, when the weights are strings, we get a "string semiring" which is $(\Sigma^* \cup \{\infty\}, \wedge, \cdot, \infty, \epsilon)$, where $\wedge$ denotes the longest common prefix operation and $\cdot$ concatenation, ∞ a new element such that for any string $w \in (\Sigma^* \cup \{\infty\}), w \wedge \infty = \infty \wedge w = w$ and $w \cdot \infty = \infty \cdot w = \infty$, defines a left semiring[3]. In fact, such a weighted automaton is precisely a "string-to-string" transducer. The languages recognized by weighted automata are called *weighted languages*. Probabilistic context-free languages (PCFG) are an example of weighted languages with the weights being probabilities. We will discuss PCFGs in detail in Section 6.

[3] A *left semiring* is a semiring that may lack right distributivity

5.2 Generalized Weight Computations in the Semiring Framework

The notation of semirings generalizes our understanding to "weights", and also enables us to take weights computations in a more general way. Typically, the weights are defined on the *paths* in the automata.

Notations Given a transition $e \in \delta$, we denote by $i[e]$ its (input) label, $w[e]$ its weight, $p[e]$ its source (or previous state) and $n[e]$ its destination state (or next state). Given a state $q \in Q$, we denote by $\delta[q]$ the set of transitions leaving q, and by $\delta^R[q]$ the set of transitions entering q.

Definition A *path* $\pi = e_1 \cdots e_k$ in A is an element of δ^* with consecutive transitions: $n[e_{i-1}] = p[e_i],\ i = 2, \ldots, k$. We extend n and p to paths by setting: $n[\pi] = n[e_k]$, and $p[\pi] = p[e_1]$. We denote by $P(q, q')$ the set of paths from q to q'. P can be extended to subsets $R \subseteq Q\ R' \subseteq Q$, by:

$$P(R, R') = \bigcup_{q \in R, q' \in R'} P(q, q')$$

The labelling function i and the weight function w can also be extended to paths by defining the label of a path as the concatenation of the labels of its constituent transitions, and the weight of a path as the $\otimes$-product of the weights of its constituent transitions:

$$i[\pi] = i[e_1] \cdots i[e_k]$$

$$W[\pi] = W[e_1] \otimes \cdots \otimes W[e_k]$$

Given a string $x \in \Sigma^*$, we denote by $\prod(x)$ the set of paths from I to F labeled with x:

$$\prod(x) = \{\pi \in P(I, F) : i[\pi] = x\}$$

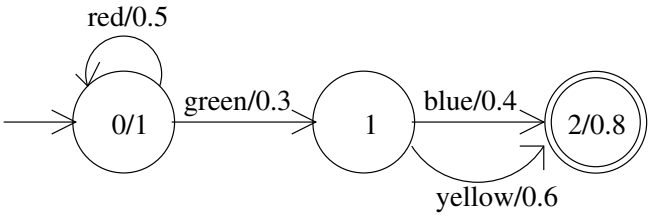
The output weight associated by A to an input string $x \in \Sigma^*$ is:

$$[\![A]\!](x) = \bigoplus_{\pi \in \prod(x)} \lambda(p[\pi]) \otimes W[\pi] \otimes \rho(n[\pi])$$

If $\prod(x) = \emptyset$, $[\![A]\!](x)$ is defined to be $\bar{0}$.

These definitions can be easily generalized to include the case of weighted automata with ϵ-transitions.

Example 5.1 Reuse the WFSA in Figure 5.1. Based on the general form defined above, we will compute two different weights by specializing the notations in the formula. First, we compute the total weight of the input $x = x_1x_2x_3$ where



$x_1 = red, x_2 = green, x_3 \in \Sigma, \Sigma = \{$red, green, blue, yellow$\}$. In this case, the semiring can be instantiated as $(R_0^1, +, \times, 0, 1)$. In the given WFSA, obviously there are two paths satisfying the given input:

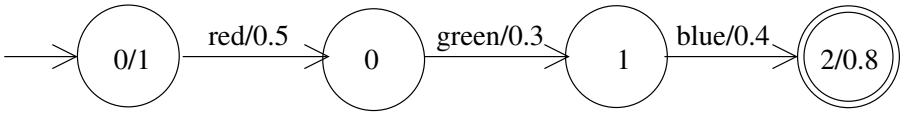


Figure 5.2: *Path π_1 for input with prefix red, green in A.*

$$\begin{aligned} W(\pi_1) &= W(e_1) \otimes W(e_2) \otimes W(e_3) \\ &= W(\delta(q_0, q_0, \text{red})) \otimes W(\delta(q_0, q_1, \text{green})) \otimes W(\delta(q_1, q_2, \text{blue})) \\ &= 0.5 \times 0.3 \times 0.4 \\ &= 0.06 \end{aligned}$$

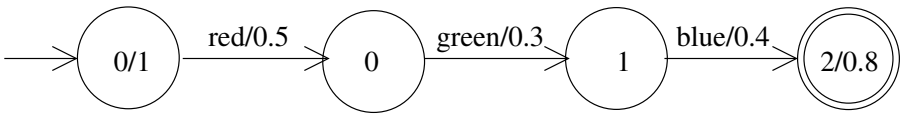


Figure 5.3: *Path π_2 for input with prefix red, green in A.*

$$
\begin{aligned}
W(\pi_2) &= W(e_1) \otimes W(e_2) \otimes W(e_3) \\
&= W(\delta(q_0, q_0, \text{red})) \otimes W(\delta(q_0, q_1, \text{green})) \otimes W(\delta(q_1, q_2, \text{yellow})) \\
&= 0.5 \times 0.3 \times 0.6 \\
&= 0.09
\end{aligned}
$$

So the total weight of input with prefix *red, green* is

$$
\begin{aligned}
[\![A]\!](x) &= \bigoplus_{\pi \in \{\pi_1, \pi_2\}} \lambda(p[\pi]) \otimes W(\pi) \otimes \rho(n[\pi]) \\
&= 1 \times W(\pi_1) \times 0.8 + 1 \times W(\pi_2) \times 0.8 \\
&= 0.06 \times 0.8 + 0.09 \times 0.8 \\
&= 0.12
\end{aligned}
$$

Similarly, if we pick the most likely path for this input, the path with the maximal weight will be chosen. In this case, the semiring can be instantiated $(R_0^1, \max, \times, 0, 1)$. We omit the same steps computing weights of π_1 and π_2. So the weight of the most likely path can be computed as follows:

$$
\begin{aligned}
[\![A]\!](x) &= \bigoplus_{\pi \in \{\pi_1, \pi_2\}} \lambda(p[\pi]) \otimes W(\pi) \otimes \rho(n[\pi]) \\
&= \max(1 \times W(\pi_1) \times 0.8, 1 \times W(\pi_2) \times 0.8) \\
&= \max(0.06 \times 0.8, 0.09 \times 0.8) \\
&= 0.072
\end{aligned}
$$

Therefore, path π_2 is the most likely path for input with prefix *red, green*.

5.3 Weighted Automata in Algebraic Automata Theory

From the last part, whenever we are talking about weighted automata, there is always some semiring involved. Moreover, the introduction of *semirings* realizes the generalization of weights computations in weighted automata. In fact, as the algebraic counterparts of weighted automata, *formal power series* play an essential role in most weighted automata applications. The weighted automaton itself is not powerful enough. More power comes from the combinations and operations over multiple automata, which is easily defined in this framework of the closure properties and operations of power series.

In this section, we first give the basic concept of formal power series, then take the weighted finite state automata (WFSA) as an example to illustrate how these algebraic concepts lay the foundations for the applications of WFSA in practice.

5.3.1 Formal Power Series

The central algebraic concepts closely connected with formal language theory are those of *monoids*, *semirings* and *formal power series* (Kuich and Salomaa, 1985; Mohri, 2002). We have given the definitions of monoids and semirings in last section.

By recalling the definition of monoids, the most important type of monoid is the *free monoid* Σ^* generated by a finite set Σ. It has all the finite strings $x_1x_2\cdots x_n$ $(x_i \in \Sigma)$ as its elements and the product $x_1 * x_2$ is formed by writing the string x_2 immediately after the string x_1, which is obviously a concatenation operation on strings. It is easy to see that a language Σ^* generated by the alphabet Σ is a free monoid in the algebraic context.

Definition A *formal power series* over Σ with *coefficients in a semiring* K is a function $S : \Sigma^* \rightarrow K$. The *image* of a word $w \in \Sigma^*$ is denoted by a pair (S, w), and is called the *coefficient* of w in S. The *support* of S is the language $supp(S) = \{w \in \Sigma^* \mid (S, w) \neq 0\}$. The set of all formal power series over Σ with coefficients in K is denoted by $K\langle\langle\Sigma^*\rangle\rangle$. The subset of $K\langle\langle\Sigma^*\rangle\rangle$ consisting of all series with a finite support is denoted by $K\langle\Sigma^*\rangle$. Series of $K\langle\Sigma^*\rangle$ are referred to *polynomials*. Generally, we are mainly interested in the polynomials and the languages over them.

Here is an example of power series. For each language $L \subset \Sigma^*$, a power series $C(L) : \Sigma^* \rightarrow K$ is defined by $(C(L), w) = 1$ if $w \in L$ and $(C(L), w) = 0$ otherwise. So semiring K is a boolean semiring, and function $C(L)$ intuitively works in a way as a "recognizer" of language L. For every word in L, it accepts it by mapping it to 1, and rejects it by mapping it to 0. Formally, such type of power series is called *characteristic series*. It can be defined over language $L \subset \Sigma^*$.

In summary, a formal power series is as a device to map a language over some semiring. Specifically, it assigns a coefficient (weight) from a semiring to each word in the language. We say that under some conditions, weighted automata is actually one way of "realizing" formal power series and reversely formal power series are the counterparts of weighted automata in algebraic automata theory. In the framework of formal power series, the notation of semirings generalizes the weight computations in weighted automata; furthermore, the closure preserving operations over power series can be applied to weighted automata, and generate more powerful automata.

5.3.2 Two Special Power Series

In this section, we simply describe two special power series – rational power series and algebraic power series. Informally, these two power series correspond to regular languages and context-free languages respectively. Therefore, regular languages sometimes are called *rational languages* and context-free languages *algebraic languages*. In the framework of power series, these two languages (power series) turn out to have some common properties and structures.

Rational Power Series and Weighted Finite State Automata The rational power series are exactly those formal power series that can be represented by finite weighted automata. We write $K^{\text{rat}}\langle\langle\Sigma^*\rangle\rangle$ for the set of rational power series over the semiring K.

Schützenberger Theorem *A formal power series in $K\langle\langle\Sigma^*\rangle\rangle$ is rational if and only if it can be represented by some weighted finite automata* (Schutzenberger, 1961).

Rational Relations and Rational Transductions A relation can be considered as a subset of the cartesian product of two sets, or as the mapping from the first set in into the set of the subset of the second. Rational relations and rational transductions describe it from the viewpoints of "static description" and "dynamic mapping" respectively, both of which are equivalent.

Definition Let Σ_1 and Σ_2 be two alphabets. A *rational relation* over Σ_1 and Σ_2 is a rational (regular) subset of the monoid $\Sigma_1^* \times \Sigma_2^*$.

Example 5.2 Suppose we have the alphabet sets $\Sigma_1 = \Sigma_2 = \{a, b\}$. We can make a rational relation $R = \{(f, g) : (f, g) \in L((a, a)+(b, b))^*(aba, aa)((a, a)+(b, b)^*)\}$ Here we use a "regular expression" constructed from a special alphabet consisting of pairs (f, g), where $f \in \Sigma_1^*$ and $g \in \Sigma_2^*$. An example of R is $(babab, baab)$.

Now we transform the "static" notion of rational ration into the "dynamic" notion of rational transduction.

A transduction τ from Σ_1^* into Σ_2^* is a function from Σ_1^* into some subset of Σ_2^*, denoted $\tau : \Sigma_1^* \to \Sigma_2^*$. The domain $\text{dom}(\tau)$ and the image $\text{im}(\tau)$ are defined

by

$$\text{dom}(\tau) = \{f \in \Sigma_1^* : \tau(f) \neq \emptyset\}$$
$$\text{im}(\tau) = \{g \in \Sigma_2^* : \exists f \in \Sigma_1^* : g \in \tau(f)\}$$

The graph of τ is the relation R defined by

$$R = \{(f, g) \in \Sigma_1^* \times \Sigma_2^* \mid g \in \tau(f)\}$$

Definition A transduction $\tau : \Sigma_1^* \to \Sigma_2^*$ is *rational* if its graph R is a rational relation over Σ_1^* and Σ_2^*.

Algebraic Power Series and Context-free Languages Let $G = (\Sigma, N, P, S)$ be a context-free grammar. If each production $X \to \alpha$ in P is associated with a weight W from some commutative semiring K, we called G is a K-weighted grammar.

Similar to the previous weight definition on "paths" in weighted automata, here we define weights on "derivation trees" based on context-free grammars. Let T be a derivation tree for $X \Rightarrow^* w$, where $X \in N$ and $w \in \Sigma^*$. Then the weight of T, denoted $W(T)$, is defined to be the product ($\otimes$) of the weights of all production occurrences in T. So the weight of input w is defined by:

$$f(w) = \bigoplus_{T:S \Rightarrow^* w} W(T) = \bigoplus_{T} \otimes_{X \to \alpha \in T} W(X \to \alpha)$$

where T ranges over all distinct derivation trees of w from S in G. If there is no derivation of w from S, then $f(w) = \bar{0}$. The function induced by the grammar, denoted f_G, is f_S.

A function $f : \Sigma^* \to K$ is called an *algebraic power series* if there is some proper[4] K-weighted grammar G such that $f = f_G$. In particular, if G is a regular grammar, then f_G is called a *rational power series*.

Closure Properties of Rational Languages, Algebraic Languages In Section 4.3, we have listed the closure properties of regular languages and context-free languages. Here we focus on the same discussion in the context of power series.

A family of languages should contain all languages having the same, or similar "structure". This structure does not depend on the alphabet chosen to represent the language. *Rat* is the family of all regular or rational languages; *Alg* is the family of context-free or algebraic languages.

[4]If every rule $X \to \alpha$ in P has either $|\alpha| \geq 2$ or $\alpha \in \Sigma$, then G is called *proper*. When G is proper, no $w \in \Sigma$ has an infinite number of derivations, and the weight function is well-defined.

Proposition 4.1 Rational Closure Properties *The families of Rat and Alg are rationally closed families. "Rationally closed" means closure under the three "rational" operations, namely union, product and the star operation.*

Proposition 4.2 Closure Properties on Rational Transductions *Each rational transduction preserves rational and algebraic languages. That is, for each rational transduction γ, $\tau(A)$ is rational if A is rational, and $\tau(A)$ is algebraic if A is algebraic.*

Definition A transduction $\tau : \Sigma_1^* \to \Sigma_1^*$ is *algebraic* if there exist an alphabet Σ^*, two morphisms $\alpha : \Sigma^* \to \Sigma_1^*$, $\beta : \Sigma^* \to \Sigma_2^*$ and a context-free language $A \subset \Sigma^*$ such that

$$\tau(f) = \beta(\alpha^{-1}(f) \cap A) \qquad f \in \Sigma_1^*$$

It follows immediately that $\tau(L)$ is context-free if $L \subset \Sigma_1^*$ is regular, and it is easy to see that $\tau(L)$ is not necessarily context-free if L is context-free.

Proposition 4.3 Closure Properties of Algebraic Transduction *Let $\tau : \Sigma_1^* \to \Sigma^*$ and $\tau' : \Sigma^* \to \Sigma_2^*$ be transductions. If one of them is rational and the other is algebraic, then $\tau' \circ \tau$ is algebraic.*

If both tranductions are algebraic, then $\tau' \circ \tau$ is not necessarily algebraic.

Here we just gave the basic facts, more formal and detailed statements can be found (Berstel, 1979).

5.4 Case Study: AT&T FSM Library^{TM}

The AT&T FSM library^{TM} is a set of general-purpose software tools available for building, combining, optimizing, and searching weighted finite-state acceptors and transducers. (M. et al., 1998) (Mohri et al., 2000) It provides a rational design of weighted finite state transducers.

The main goal of the library is to provide algorithms and representations for all the symbolic processing components (phonetic, lexical, and language-modeling) in large-vocabulary speech recognition systems. This imposed the following requirements:

- Generality: to support the representation and use of the various information sources in speech recognition
- Modularity: to allow rapid experimentation with different representations
- Efficiency: to support competitive large-vocabulary recognition using automata of more than 10 million states and transitions.

One of the central steps of the library design is to factor the task under study into algorithm and data structures. A mathematical analogue is suggested here that the separation of algebra and algorithms. In other words, the library should be designed to work in as general an algebraic structure as possible.

By taking advantage of the theory of rational power series and semirings, which supply the semantics for the objects and operations, a high degree of generality, modularity and irredundancy are achieved in this toolkit. Also, the generalization based on rational power series creates opportunity for optimizations (such as on-demand composition, determinization and minimization) that are not available in more "ad hoc" speech recognition frameworks.

5.4.1 Semirings in FSM Toolkit

Previously we have given a general form to compute the weights for a particular input in weighted automata based on the definitions of semirings and paths. The same form can be directly applied to the WFST case. Most of the algorithms of FSM library can work with arbitrary semirings or with semirings from properly-defined subclasses. For instance, some algorithms require *commutative semirings*; others require *closed semirings*, in which an infinite addition operation is defined and behaves like a finite addition with respect to multiplication operation. To instantiate the library for a particular semiring K, we just need to give computational representations for the semiring elements and operations.

For example, the same power series determinization algorithm and code can be used to determinize transducers, weighted transducers, weighted automata. To do so, one just needs to use the algorithm with the string semiring $(\Sigma^* \cup \{\infty\}, \wedge, \cdot, \infty, \epsilon)$ in the case of transducers, with the semiring $(R, +, \cdot, 0, 1)$ and $(R_+, \min, +, \infty, 0)$ in the other cases, and with the cross product of the string semiring and one of these semirings in the case of weighted transducers. Similarly, a general single-source shortest-paths algorithm can be used to compute the single-source shortest distance when weights are numbers, strings, or even sets. Refer to (Mohri, 2002) for further details.

5.4.2 Rational Operations on Weighted Transducers

In the FSM library, the rational operations (union, concatenation, Kleene closure), reversal, inversion and projection operations and composition operations (namely transducer composition, acceptor intersection) are implemented. When taking these operations over multiple weighted finite transducers and acceptors, a closure property has to be guaranteed, which means the resulting automaton still retains the original properties, and the resulting semiring still retains the original structure. Rational power series lay the theoretical foundation for the design of FSM library, therefore support all these well-defined operations in the library. Furthermore, the cascade of weighted transducers and acceptors provides a more flexible and powerful toolkit.

5.5 Summary and Open Problems

In this section, we explored the algebraic foundation of weighted language and weighted automata. Putting (weighted) languages and automata into the framework of formal power series gives us another perspective of looking at languages. In particular, the notation of semirings generalizes the representations and computations of weights for weighted languages and power series itself provides us another way to generate languages and the coefficients for languages. As a successful application of rational power series, AT&T FSM library motivates us to do more explorations with this perspective, such as how to exploit the properties of the algebraic power series on context-free languages, how to take advantage of the closure properties of rational transductions on context-free languages and how to formalize various tasks and methods (such as learning methods) dealing with context-free languages into a proper framework of power series and semirings. Furthermore, studying and exploring the algebraic properties for other tractable language formalisms is still open.

6 Context-free Parsing algorithms

There exist efficient recognition and parsing algorithms for arbitrary context-free languages. Compared with a pushdown automaton which recognizes a particular context-free language, these algorithms are more general and can be used for any context-free language; And there is a correspondence between these polynomial algorithms and algebraic power series: *Any algebraic power series* $f : \Sigma^* \rightarrow K$

can be evaluated at string t in Σ^* in polynomial time. In this section, we are going to introduce some fundamental recognition and parsing algorithms, and how they work for one particular weighted context-free language – probabilistic context-free languages.

6.1 Recognition and Parsing Algorithms for Context-free Languages

The recognition problem for context-free languages is essentially a decision problem: "Given a context-free grammar G and a string w, is $w \in L(G)$"? A parsing process is basically a recognition process which additionally outputs a parse or derivation of each acceptable input. So if we can effectively recognize a string, then the parsing is trivial. Once recognition is accomplished, we can use backtracking strategy to retrieve the derivations in linear time.

In the following, we give three classic context-free parsing algorithms: CKY algorithm, Earley's algorithm and GHR parsing algorithm.

6.1.1 CKY (Cocke-Kasami-Younger) Parsing Algorithm

The CKY algorithm is a tabular dynamic programming method. It works bottom-up by finding all derivations for a string by using previously computed derivations for all possible substrings. The remarkable property of CKY algorithm is that it has the ability to parse arbitrarily ambiguous CFGs and find all possible parse trees (exponentially many) for a given input in polynomial time and space. As we will see, its time complexity is $O(|G|^2 n^3)$ and its space complexity is $O(|G|n^2)$ where $|G|$ is the size of grammar and n is the length of input string.

Figure 6.1 shows the basic CKY algorithm:

CKY Algorithm **Input string** w **of size** n
Initialize $2D$ **chart of size** n^2 //each cell of chart can be viewed as a set
for $i = 0$ **to** $n - 1$
add A into chart$[i][i+1]$ **if** $w[i] = a$ **and** rule $A \rightarrow a \in G$
for $j = 2$ **to** N
  for $i = j - 2$ **downto 0**
    for $k = i + 1$ **to** $j - 1$
    add A into chart$[i][j]$ **if** $B \in$ chart$[i][k]$ **and** $C \in$ chart$[k][j]$ **and** rule

$A \rightarrow BC \in G$
return "yes" **if** $S \in$ chart$[0][n]$
else return "no"

Figure 6.1: CKY algorithm

Limitations of CKY There are some limitations in practice when we use the CKY algorithm.

- The given grammar has to be transformed into Chomsky Normal Form which typically will increase the size of the grammar; the transformation has some extra time cost and the derivations are from the transformed grammar rather than from the original grammars.
- It runs in time $O(|G|^2 n^3)$. In many applications $|G|$ is much bigger than n, so squaring the size of the grammar can have a drastic effect on performance.
- It may spend lots of time making "useless" matches. That is, it finds every variable A matching some substring $w_{i,j}$ without regard to whether or not that match can occur within the context of the rest of the sentence, i.e., whether or not $S \Rightarrow^* w_{0,i} A w_{j,n}$. Matches which fail to satisfy this criterion may make up the great majority of all matches found.

See Section 7.1.2 for more on CKY parsing.

6.1.2 Earley's Parsing Algorithm

Earley's algorithm is proposed by (Earley, 1970). It is a tabular parsing method based on a dynamic programming strategy. However, it avoids the main limitation of CKY algorithm and applicable to arbitrary context-free grammars without relying on any grammar transformations. It uses a top-down parsing strategy with a bottom-up filter in a way that it initializes the top-down rule in form of $S \rightarrow \alpha$ and scan the string from left to right with applying three kinds of rules: *predictor, scanner* and *completer*. Earley algorithm uses "lookahead" by using "dotted rules" to avoid lots of useless matchings, which is superior to CKY again. In some cases, lookahead may lead to things more complicated and increases the number of states. We will not explain this algorithm in full length. See Section 7.1.2 for more of this algorithm.

6.1.3 GHR (Graham-Harrison-Ruzzo) Algorithm

GHR (Graham-Harrison-Ruzzo) algorithm (L.Graham et al., 1980) is another improved recognition algorithm working on arbitrary context-free grammars. As we know, the principle drawback of the CKY method is that the algorithm may find a lot of "useles" matches which cannot lead to derivations to S. Compared with CKY, GHR algorithm improves the average case performance by considering only those matches that are consistent with the left context, which is basically as same as Earley algorithm does. Specifically:

CKY : *Add* A *to chart*$[i, j]$ *if and only if* $A \Rightarrow^* w_{i,j}$
GHR : *Add* $A \rightarrow \alpha \cdot \beta$ *to chart*$[i, j]$ *if and only if* $\alpha \Rightarrow^* w_{i,j}$ *and* $S \Rightarrow^* w_i A\delta$ *for some* $\delta \in \Sigma^*$.

Another trick in GHR is to pre-compute the "prediction set", which saves time by such off-line computation. GHR seems to do as well as one could expect in an *on-line* recognizer that recognizes each prefix of w without lookahead. Still the algorithm runs in time $O(|G^2|n^3)$ and space $O(|G|n^2)$ for arbitrary context-free grammar. And it is also possible to implement the algorithm in time asymptotically less than $O(|G^2|n^3)$.

6.1.4 Summary of CKY, Earley's and GHR Algorithms

The algorithms of CKY, Earley and GHR are different styles of parsing for context-free grammars (CFGs). All of them ensure that, given a set of grammar rules and an input sentence, all possible parses of that sentence are produced. The dynamic programming strategy makes the results not only include all the possible parse trees but all of the subtrees that were found during the generation of complete parse trees. In particular, there are some close connections between CKY and Earley, which is shown by relating both to GHR ((L.Graham et al., 1980) section 5). The unification of superficially dissimilar methods may give us some clue to represent these different-appearance algorithms in a more general way. It turns out it is feasible to achieve. We put this dicussion in Section 7.

6.2 Probabilistic Context-free Grammars (PCFGs)

From the previous discussion, we know that ambiguity is a basic problem in computational linguistics. One solution is by using *probability* to help find the best parse tree among the potentially exponentially many parses, say in the parsing

task. Probabilistic context free grammar is proposed based on this motivation. And as one particular weighted context-free grammar, it can be used to study weighted context-free parsing.

Definition A *probabilistic context-free grammar* (or PCFG) is a context-free grammar that associates a probability to each of its production rules. Formally, a PCFG G consists of (Σ, N, P, S) and a corresponding set of probabilities on rules such that:

$$\forall N^i \in N \quad \sum_{\alpha} P(N^i \to \alpha) = 1$$

The following grammar is a well-defined probabilistic context-free grammar:

$$\begin{array}{lcll} S & \to & XX & 0.7 \\ S & \to & YY & 0.3 \\ X & \to & AA & 0.4 \\ X & \to & B & 0.6 \\ Y & \to & AA & 0.5 \\ Y & \to & B & 0.5 \\ A & \to & a & 0.1 \\ B & \to & b & 0.1 \end{array}$$

A PCFG generates the same set of parses for a given input that the corresponding CFG does, and assigns a probability to each parse tree. The probability of a parse tree is the product of the probabilities of all rules applied in the parse tree. The probability of a sentence is the sum of the probabilities of all parse trees.

Example 6.2 We use the above grammar to parse the input aab, and get two parse trees:
The probabilities for these two trees are:

$$\begin{aligned} P(T_1) &= P(S \to XX) \times P(X \to AA) \times P(X \to B) \times P(A \to a)^2 \times P(B \to b) \\ &= 0.7 \times 0.4 \times 0.6 \times 0.1^2 \times 0.1 = 0.000168 \\ P(T_2) &= P(S \to YY) \times P(Y \to AA) \times P(Y \to B) \times P(A \to a)^2 \times P(B \to b) \\ &= 0.3 \times 0.5 \times 0.5 \times 0.1^2 \times 0.1 = 0.000075 \end{aligned}$$

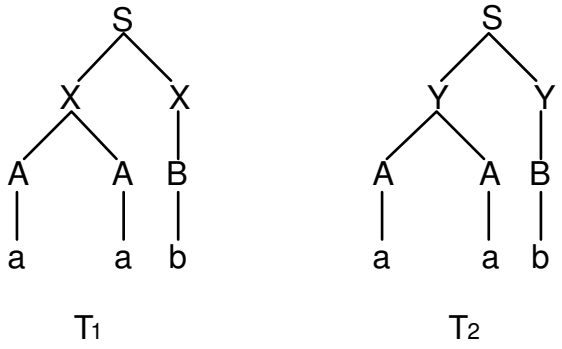


Figure 6.1: *parse trees of* aab.

Since T_1 and T_2 are all possible parse trees for input aab, then the entire probability of generating input aab is : $P(aab) = P(T_1) + P(T_2) = 0.000168 + 0.000075 = 0.000243$. Obviously, between t_1 and T_2, T_1 is the more likely parse tree for input aab based on the given grammar.

6.2.1 Inside Algorithm and Viterbi Algorithm

As illustrated in the example, generally we are more interested in two types of probabilities based on PCFGs:

- What is the probability of a sentence w_{1m} according to the grammar G: $P(w_{1m} \mid G)$?
- What is the most likely parse tree for a sentence: $\text{argmax}_t P(t \mid w_{1m}, G)$?

Typically, the first probability is called *inside probability*[5] of the input sentence; and the second *Viterbi probability*[6] for the input sentence.

[5]Formally, the inside probability $\beta_j(p,q)$, $\beta_j(p,q) = P(w_{pq} \mid N^j_{pq}, G)$, is the total probability of generating words $w_p \cdots w_q$ given that one is starting off with the nonterminal N^j.

[6]By the notation $\delta_i(p,q) =$ the highest inside probability parse of a subtree N^i_{pq}, the Viterbi probability for generating words $w_p \cdots w_q$ starting off with the nonterminal N^i is $\delta_i(p,q) = \max_{1 \le j,k \le n, p \le r < q} P(N^i \rightarrow N^j N^k) \delta_j(p,r) \delta_k(r+1,q)$

Intuitively, if we simply enumerate all possible parse trees of the string, just as the example illustrated, the answers to both probabilities are pretty straightforward. However, in general it is impractical as there might be exponentially many number of parse trees. We have to explore a more efficient way to calculate those values. Inside algorithm and Viterbi algorithm use dynamic programming to compute the inside probability and the Viterbi probability respectively.

The following are the CKY-style inside algorithm and Viterbi algorithm for PCFGs:

CKY-style Inside Algorithm float chart$[0..n-1,\ 1..|N|,\ 0..n] = 0$
for $i = 0$ **to** $n-1$
  for each rule $A \rightarrow w_s \in P$
    chart$[i, A, i+1] = P(A \rightarrow w_s)$
for $j = 2$ **to** N
  for $i = j-2$ **downto 0**
    for $k = i+1$ **to** $j-1$
      for each rule $A \rightarrow BC \in P$
        chart$[i, A, j]$ = chart$[i, A, j]$+(chart$[i, B, k]\times$ chart$[k, C, j] \times P(A \rightarrow BC)$
return chart$[0, S, n]$

Figure 6.2: CKY-style Inside Algorithm

CKY-style Viterbi Algorithm
float chart$[0..n-1, 1..|N|, 0..n] = 0$
for $i = 0$ **to** $n-1$
  for each rule $A \rightarrow w_s \in P$
    chart$[i, A, i+1] = P(A \rightarrow w_s)$
for $j = 2$ **to** N
  for $i = j-2$ **downto 0**
    for $k = i+1$ **to** $j-1$
      for each rule $A \rightarrow BC \in P$
        chart$[i, A, j]$ = max(chart$[i, A, j]$,(chart$[i, B, k]\times$ chart$[k, C, j]\times P(A \rightarrow BC)$)
return chart$[0, S, n]$

Figure 6.3: CKY-style Viterbi Algorithm

Notes and Comments:

1. Here we use the form of chart$[i, A, j]$ to denote different element in the same cell of the table, which is different from the notation in basic CKY algorithm.

2. Comparing these two algorithms by observation, Viterbi algorithm finds the most probable parse by taking the maximum operation instead of summing over all parse trees as in inside algorithm. The "surprising" similarity between these two algorithms are essentially consistent with the previous discussion of semirings and algebraic power series in Section 5. We can use the notation of semirings to unify these two algorithms. And Inside algorithm corresponds to the semiring $(R_0^1, +, \times, 0, 1)$ and Viterbi algorithm corresponds to the semiring $(R_0^1, \max, \times, 0, 1)$.

We can rewrite these two algorithms into one general form as follows:

chart$[0..n-1, 1..|N|, 0..n] = \bar{0}$
for $i = 0$ **to** $n - 1$
  for each rule $A \rightarrow w_s \in P$
    chart$[i, A, i+1] = P(A \rightarrow w_s)$
for $j = 2$ **to** N
  for $i = j - 2$ **downto 0**
    for $k = i + 1$ **to** $j - 1$
      for each rule $A \rightarrow BC \in P$
        chart$[i, A, j] = \bigoplus$(chart$[i, A, j]$,chart$[i, B, k] \otimes$ chart$[k, C, j] \otimes P(A \rightarrow BC)$)
return chart$[0, S, n]$

Figure 6.4: A generic CKY-style Algorithm

Similarly, we can use a generic Earley or GHR parsing algorithm to compute both probabilities; furthermore, we can compute top-n most likely parse trees for a given input using the same generic algorithm. All the computations can be generalized into the same framework based on the concept of power series and the notation of semirings. We will discuss it in full length in Section 8.

6.2.2 Assumptions underlying PCFGs

Notice that when we compute the probabilities for the substrings, there are some assumptions we are implicitly using:

- **Place invariance:** the probability of a subtree does not depend on where in the string the words it dominates are (cf. **time invariance** in HMMs):

$$\forall k \quad P(N^j_{k(k+c)}) \text{ is the same}$$

- **Context-free:** the probability of a subtree does not depend on words not dominated by the subtree:

$$P(N^j_{kl} \rightarrow \zeta | \text{ anything outside } k \text{ through } l) = P(N^j_{kl} \rightarrow \zeta)$$

- **Ancestor-free:** the probability of a subtree does not depend on the nodes in the derivation outside the subtree:

$$P(N^j_{kl} \rightarrow \zeta | \text{ any ancestor node outside } N^j_{kl} = P(N^j_{kl} \rightarrow \zeta)$$

From the above analysis, PCFGs is the simplest and most natural probabilistic model for tree structure, the algorithms for them are a natural development of the algorithm employed with Hidden Markov Models (HMMs), and PCFGs provide a sufficiently general computational device that they can simulate various other forms of probabilistic conditioning. Nevertheless, it is important to realize that PCFGs are only one of many ways of building probabilistic models of syntactic structures.

6.3 Summary of this Section

In this section, we introduced the basic parsing algorithms of context free grammars (CFGs) and probabilistic context free grammars (PCFGs). Parsing algorithms play a core role both in the syntactic structure analysis and value computations related to syntactic structures. Combining the semiring notation, parsing algorithms across tasks can be unified into a general form, which rejustifies the statements in Section 5.

7 Deductive Parsing

From the previous discussion, we see that the very same parsing algorithm can be applied across various tasks over different semirings. From the insights provided by GHR parsing, which indicates the close connections between CKY, Earley and GHR parsing algorithms and other parsing strategies, we may ask if there is a unified way to represent all these different parsing strategies. Deductive parsing (S. M. Shieber, 1995) provides us such a general framework by taking parsing as a deductive process in a way that seeks to prove claims about the grammatical status of a string from assumptions describing the grammatical properties of the string's elements and the linear order between them. One insight and technique to the study of grammar formalisms and parsing is that the modular separation of parsing into a logic of grammaticality claims and a proof search procedure allows the investigation of a wide range of parsing algorithms for existing grammar formalisms by selecting specific classes of grammaticality claims and specific search procedures.

7.1 Components of a Deductive System

A deductive parsing system consists of an *item-based description* component and a *control engine* component. Item-based descriptions are basically used to describe parsing algorithms by treating parsing as deduction, and the control engine component interprets the deduction to implement the corresponding parser. In the following, we will introduce these components and use some examples to illustrate the idea of deduction.

7.1.1 Item-based Description: Items and Inference Rules

In most parsers, there is at least one chart of some form. In this general description, a corresponding concept *item* is proposed with the form $[i,A,j]$. For example, in CKY parsing, item is of the form $[i,A,j]$ and states that the nonterminal A derives the substring between index i and j in the string. We use this form to denote every chart element.

The general form for an inference rule will be

$$\frac{A_1 \cdots A_k}{B} \langle \text{side conditions on } A_1 \cdots A_k, B \rangle$$

where A_1,...,A_k the *antecedents* and B the *consequent* of the inference rule. The

existence of side conditions depends on the parser and its value is only true or false. When the inference rule is used, the metavariables it contains will be instantiated by appropriate *terms*. *Terms* refer to grammar rules and items.

An *axiom* in a deductive parsing refers to a sound item. For example, for each word w_i in the string, it is clear that the item [i,w_i,$i+1$] makes a true claim, so that such item can be taken as axiomatic. Generally we start our parsing procedure with axioms.

7.1.2 Item-based Description: Examples of CKY and Earley Parsing

The following gives item-based descriptions of a CKY-style parser and Earley-style parser respectively.

Item form: [i,A,j]

Goal: [0,S,n]

Axioms: [i,w_i,$i+1$] $\quad 0 \leq i < n$

Inference rules: $\frac{R(A \rightarrow w_i)}{[i,A,i+1]}\,[i, w_i, i+1]$ Prediction

$\frac{R(A \rightarrow BC) \quad [i,B,k] \quad [k,C,j]}{[i,A,j]}$ Completion

Figure 7.1 *Item-based description of CKY parser*

Item form: [i,$A \rightarrow \alpha \bullet \beta$,$j$]

Goal: [0,$S' \rightarrow S\bullet$,n]

Axioms: $[0, S' \rightarrow \bullet S, 0]$

Inference rules: $\frac{[i,A \rightarrow \alpha \bullet w_j \beta, j]}{[i,A \rightarrow \alpha w_j \bullet \beta, j+1]}$ Scanning

$\frac{R(B \rightarrow \gamma)}{[j,B \rightarrow \bullet \gamma, j]}[i, A \rightarrow \alpha \bullet B\beta, j]$ Prediction

$\frac{[i,A \rightarrow \alpha \bullet B\beta, k] \quad [k,B \rightarrow \gamma \bullet, j]}{[i,A \rightarrow \alpha B \bullet \beta, j]}$ Completion

Figure 7.2 *Item-based description of Earley parser*

In this representation, we use some conventional notations for metavariables ranging over the objects under discussion: n for the length of the object language string to be parsed; $A, B, C, \ldots$ for arbitrary formulas or symbols such as grammar nonterminals; $a, b, c, \ldots$ for arbitrary terminal symbols; $i, j, k, \ldots$ for indices into various strings, especially the string w; $\alpha, \beta, \gamma, \ldots$ for strings or terminal and nonterminal symbols.

Here we use context-free grammars (CFGs) to illustrate the description format even though it can similarly be extended to other formalisms.

7.1.3 Interpretation Engine – Control Strategy

The specification of inference rules only partially characterizes a parsing algorithm, in that it provides for what items are to be computed, but not the order. So a further control strategy is needed to operate over the inference rules. One crucial concern is that we do not want to enumerate an item more than once. To prevent this, it is standard to maintain a cache of lemmas, adding to the cache only those items that have not been seen so far. This cache plays the same role as the *chart* in the chart-parsing algorithms, the *well-formed substring table* in CYK parsing, and the *state sets* in Earley's algorithm.

As a reasoning engine, the soundness and completeness of this strategy should be proved when in use.

In general, item should be added to the chart as they are proved. However, each new item may itself generate new consequences. The issue as to when to compute the consequences of a new item is quite subtle. So a standard solution is to keep a separate *agenda* of items that have been proved but whose consequences have not been computed. When an item is removed from the agenda and added to the chart, its consequences are computed and themselves added to the agenda for later consideration.

7.2 Item-based Descriptions

Item-based descriptions in deductive parsing provides us a facility to represent a variety of parsing algorithms in a unified way. An item-based description provides the flexibility and generality of the deductive parsing. Some related work are listed as follows:

7.2.1 Prolog Implementation of Deductive Parsing

In (S. M. Shieber, 1995), a deductive parsing system is implemented in Prolog. In this system, a series of parsing algorithms are literally represented by *inference rules*, and a uniform *deduction engine* is given by parameterized with the inference rules so that it can be used to parse according to any of the associated algorithms.

7.2.2 Dyna

Dyna (*http://www.dyna.org*)is a Turing-complete programming language for specifying dynamic programs and training their weights. You write a short declarative specification in Dyna, and the Dyna optimizing compiler produces efficient C++ classes that form the core of your C++ application. For example, for CKY inside parsing algorithm

:- item(item, double, 0).
constit(X, I, J) += rewrite(X, W) $*$ word(W, I, J).
constit(X, I, K) += rewrite(X, Y, Z)$*$constit(Y, I, J)$*$ constit(Z, J, K).
goal += constit($s, 0, N$) $*$ end(N).

Dyna has a similar form with Prolog. It can express the abstract structure of an algorithm.

7.3 Summary of this section

Deductive parsing provides us a general way to represent different parsing algorithms. By taking parsing as deduction, it uses item-based description to describe the parsing algorithm directly and a single control strategy to implement the corresponding parser. The modularity of deductive parsing is especially useful for rapid prototyping of an experimentation with new parsing algorithms. It can be easily extended to develop algorithms for parsing with tree-adjoining grammars, combinatory categorial grammars, and also lexicalized context-free grammars. A wider perspective is to extend the deductive parsing strategy to many other NLP applications. Specifically, the notation of item-based description provides us a hint that a lot of mapping/deductive relations can be represented in this fashion, which gives more space to explore the deductive system on various NLP applications, such as the mapping between bilingual texts for machine translation, grammar transformations, etc. If we can formalize the problem of interest as a deduction

process, then we can use this general item-based description. We will take it as one of our future work.

8 Semiring Parsing

From the discussion before, we know that a variety of parsing algorithms can be represented as a deductive procedure based on the generalized item-based descriptions. As we have seen, parsing algorithms can be used to compute many interesting quantities in practice, such as *Viterbi value, Viterbi derivation, n-best derivations, derivation forests, derivation count, inside and outside probabilities* of components of the string. Traditionally, a different parser description is needed to compute each of these values; a fair amount of work can be required to construct each one. Therefore it is preferable to unify all these parsers into a general framework to make it work across tasks and fields. Since we already have a unified deductive way to represent different parsing algorithms. And actually parsing has solid algebraic foundation in the *formal power series* framework, which we have discussed in detail in previous sections. All these provide a probability of unifying statistical NLP into a general framework by taking advantage of the algebraic properties of parsing algorithms. A semiring parsing framework is proposed and studied in (Goodman, 1999).

8.1 Principles of Semiring Parsing

In previous section, we talked in detail about that parsing can be brought into an algebraic power series framework. It generalizes the notion of recognition, parsing, statistical parsing, etc. by describing the computations executed from an input to a domain with a semiring structure. An abstract semiring plays a central role in a power series framework on related domain of grammars and parses. In algebraic power series framework, each production of G is associated with an element of a semiring A, and parsing amounts to finding the element of A associated with x (the *coefficient* of x) with respect to the grammar. When one deals with the boolean semiring, the power series represents recognition; when one uses the semiring of parse forests, the power series represents parsing.

Semiring parsing system is essentially an implementation of an algebraic power series framework of parsing. It associates the grammar with specific semiring in terms of particular task. In addition, it simplifies the representations of a variety of parsers with a unified *item-based description*; in particular, such unification

makes it easier to generalize parsers across tasks: a single item-based description can be used to compute values for a variety of applications, simply by changing semirings.

In the following part, we will use examples to make the discussion more transparent.

8.2 Examples: Inside Semiring and Viterbi Semiring

A semiring parser requires a grammar, a string to be parsed and an item-based description of the parsing algorithm as input. In this section, we will use an example to illustrate how semiring parsing works. For simplicity, we will use CKY parsing algorithm working on Inside semiring and Viterbi semiring.

The Viterbi semiring computes the probability of the most probable derivation of the input sentence, given a probabilistic grammar. It is formalized as: $(\mathrm{R}_0^1, \max, \times, 0, 1)$. R_0^1 indicates the set of real numbers from 0 to 1 inclusive, which denotes the domain of grammar rules and parses values. And $\oplus$ in abstract semiring is specialized into *max*, $\otimes$ into $\times$, identity $\bar{0}$ into 0 and identity $\bar{1}$ into 1 with general semantics. Since we handle *probabilities* in this task, it is easy to understand the definition of the semiring. Similarly, inside semiring is $(\mathrm{R}_0^1, +, \times, 0, 1)$.

The input string is xxx and the grammar is as follows:

$$\begin{array}{lcll} S & \rightarrow & XX & 1.0 \\ X & \rightarrow & XX & 0.2 \\ X & \rightarrow & x & 0.8 \end{array}$$

We use item-based description of CKY parser described before.

Starting from axioms, which are $[0, x, 1], [1, x, 2], [2, x, 3]$ respectively, prediction inference rule is firstly triggered and instantiated as

$$\frac{R(X \rightarrow x)}{[0,X,1]}[0, x, 1], \quad \frac{R(X \rightarrow x)}{[1,X,2]}[1, x, 2], \quad \frac{R(X \rightarrow x)}{[2,X,3]}[2, x, 3].$$

Axioms are obviously true, so we compute the following items values:

$$\begin{aligned}[0, X, 1] &= 0.8\\ [1, X, 2] &= 0.8\\ [2, X, 3] &= 0.8\end{aligned}$$

Next, completion inference rule can be used. Consider the instantiation $i = 0, k = 1, j = 2, A = X, B = X, C = X$:

$$\frac{R(X \rightarrow XX) \quad [0, X, 1] \quad [1, X, 2]}{[0, X, 2]}$$

We use the multiplicative operator of the Viterbi semiring to multiply together the values of the top line, deducing that $[0, X, 3] = 0.2 \times 0.8 \times 0.8 = 0.128$. Similarly,

$$\begin{aligned}[0, X, 2] &= 0.128\\ [1, X, 3] &= 0.128\\ [0, S, 2] &= 0.64\\ [1, S, 3] &= 0.64\end{aligned}$$

There are two more ways to instantiate the conditions of the completion rule:

$$\frac{R(S \rightarrow XX) \quad [0,X,1] \quad [1,X,3]}{[0,S,3]}$$

$$\frac{R(S \rightarrow XX) \quad [0,X,2] \quad [2,X,3]}{[0,S,3]}$$

The first has the value of $1 \times 0.8 \times 0.128 = 0.1024$, and the second has the value 0.1024. For the ambiguities, we typically use additive operator of semiring to "sum" them up. In particular, for Viterbi semiring, additive operator is *max*. So the Viterbi value for goal item is $[0, S, 3] = \max(0.1024, 0.1024) = 0.1024$. For inside semiring, inside value for goal item is $[0, S, 3] = 0.1024 + 0.1024 = 0.2048$.

Practically, we use semiring to compute the item values. There are some important detail for semiring parser we cannot skip – the order of item value computation. Specifically, we cannot compute the value of a consequent item until the

values of all its antecedents have been computed. Basically, if we can give an order on all derivable items, we can compute the value of item one by one according to this order. In our system this method turns out feasible but suboptimal in space complexity. Section 4.3 sees more discussions.

As for the proof of correctness of semiring parsing and more detailed discussion, refer to (Goodman, 1999).

8.3 Some Classic Semirings in Semiring Parsing

The following are some classic semirings in semiring parsing. We are not going to discuss them in detail.

Boolean Semiring $(\text{TRUE, FALSE}, \vee, \wedge, \text{FALSE, TRUE})$ recognition

Inside Semiring $(\mathrm{R}_0^1, +, \times, 0, 1)$ string probability

Counting Semiring $(\mathrm{N}_0^\infty, +, \times, 0, 1)$ number of derivations

Derivation Forests Semiring $(2^E, \cup, \cdot, \emptyset, \{\langle\rangle\})$ set of derivations

Viterbi Derivation Semiring
$(\mathrm{R}_0^1 \times 2^E, \max_{Vit}, \times_{Vit}, \langle 0, \emptyset\rangle, \langle 1, \{\langle\rangle\}\rangle)$ best derivation

Viterbi-n-best Derivation Semiring
$(\left\{\text{topn}(X) | X \in 2^{R_0^1 \times E}\right\}, \max_{Vit-n}, \times_{Vit-n}, \emptyset, \langle 1, \{\langle\rangle\}\rangle)$ best n derivations

E is the set of all derivations in some canonical form, and 2^E to indicate the set of all sets of derivations in canonical form.

8.4 Semiring Parsing – A Prototype Implementation

Based on the description of (Goodman, 1999), a preliminary semiring parsing framework has been implemented. It is implemented in ANSI/ISO standard C++ and using STL methods and functions. We experiemented the Viterbi semiring and inside semiring on CKY and Earley parsing. From perspective of functionality, the working system can be divided into *item generation* part and *item value computation* part; from perspective of design, the overall working framework can be divided into four components:

- Item-based description component,
- Graph generation and sorting component,
- Interpreter component,
- Item value computation and update component.

Here we mainly talk about the system from the latter respect. Figure 8.1 shows the communications between the components. In the following sections, we will describe each component in detail.

8.4.1 Item-based Description Component

In last section, an *item-based description* has been exemplified with CKY parser. In a working system, this component is initialized by a particular parsing algorithm, which is described in such a generalized form. Especially, user can customize his own parser according to this form and initialize the component in his own right, which adds more merits of flexibility for the system. Essentially this component takes a *deductive parsing-style* strategy and the inference rules play a central role in it.

8.4.2 Graph Generation and Sorting Component

It has been mentioned in section 8.2 that we need to impose an ordering on the items, in such a way that no item precedes any item on which it depends. To this end, each item x will be assigned to a *bucket* B, writing *bucket*$(x) = B$ and saying x is *associated* with B. So buckets are ordered in such a way that if item y depends on item x, then *bucket*$(x) \leq$ *bucket*(y). For those items that depend on themselves or depend on each other, they are associated with special *looping buckets*, whose value may require infinite sums to compute. For simplicity, we will leave out looping case for the moment in our discussion.

Overview of Bucket Sorting One way to organize the buckets with the required ordering constraints is to create a graph of the dependencies, with a node for each item, and an edge from each item x to each item b that depends on it. Then we can separate the graph into its strongly connected components (which are associated with looping buckets), and perform a topological sort. Since we do not consider looping case for now, we suppose every bucket associate only one item.

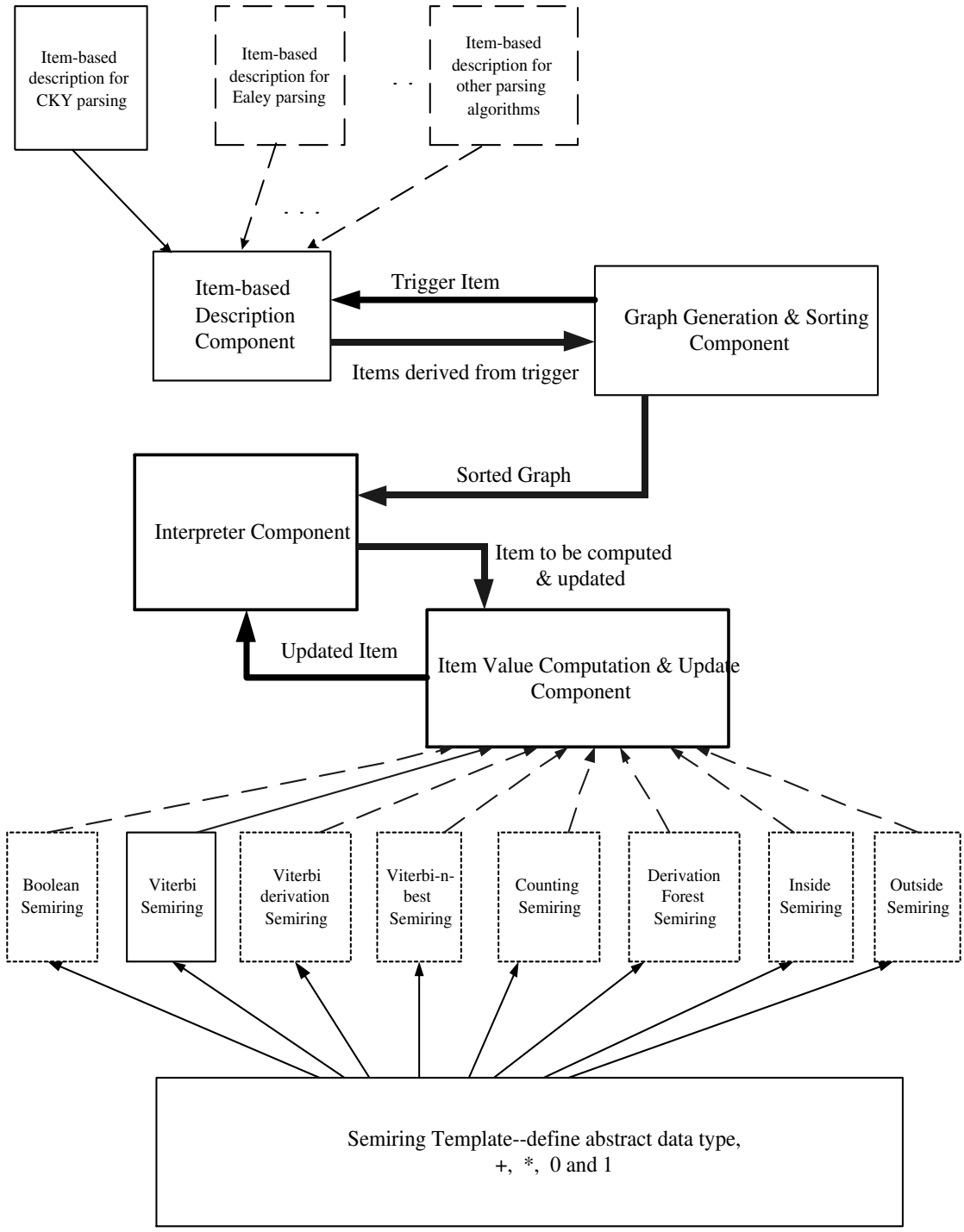


Figure 8.1: *Semiring Parsing System Framework*

Generally, there are three approaches to determine the buckets and ordering.

The first one is simply a brute-force enumeration of all items, derivable or not, followed by a topological sort. Obviously, this approach will have suboptimal time and space complexity for most item-based description.

The second approach is to use a deductive parser to determine the derivable items and their dependencies, and then to perform a topological sort. Theoretically, the time complexity of deductive parsing can achieve optimal ($O(n^3)$) and the time complexity of topological sort is bounded by $O(|V| + |E|)$, where $|V|$ is the number of items that can be bounded by $O(n^2)$ and E is the number of dependencies of the graph, which is bounded by a cubic complexity (Billot and Lang, 1989). So the entire time complexity can achieve optimal; Unfortunately, for memory, there are several $O(E)$ data structures holding edges. Therefore, the space complexity of this approach will not achieve $O(n^2)$, which denotes optimal space complexity.

The third approach to bucketing is to create algorithm-specific bucketing code; this results in a parser with both optimal time and optimal space complexity. For instance, in a CKY-style parser, we can simply create one bucket for all items with same span of length. Essentially, to achieve optimal performance, this approach is trying to put the items with no any dependencies in between into the same bucket. It helps maximize sharing of structure and reduce the number of dependencies, which is a key factor in the space complexity.

Example of the Second Approach In our system, we implement the second approach in *graph generation and sorting component*. In this subsection, we will still use the example in Section 8.2 to illustrate this process. Figure 8.2 shows the dependency graph generated by approach 2. In this graph, each node denotes a derived item and a directed edge from node i to node j denotes the dependency relation of item i to item j. The graph is an *AND-OR graph*; *AND-nodes* correspond to the usual derivable items, which can be derived from some unique set of antecedents. For instance, all nodes in the graph but $[0, X, 3]$ and $[0, S, 3]$ are AND-nodes; while *OR-nodes* correspond to those items that can be derived from multiple sets of antecedents, i.e. such items can be deducted from different paths that represent ambiguities. The nodes $[0, X, 3]$ and $[0, S, 3]$ in the graph are the examples of OR-nodes, which have two different derivation paths respectively.

Note that in this graph every item is derivable and associated with one bucket implicitly. A *chart-based, agenda-driven* strategy is used to control the item derivation by iteratively invoking the item-based description component.

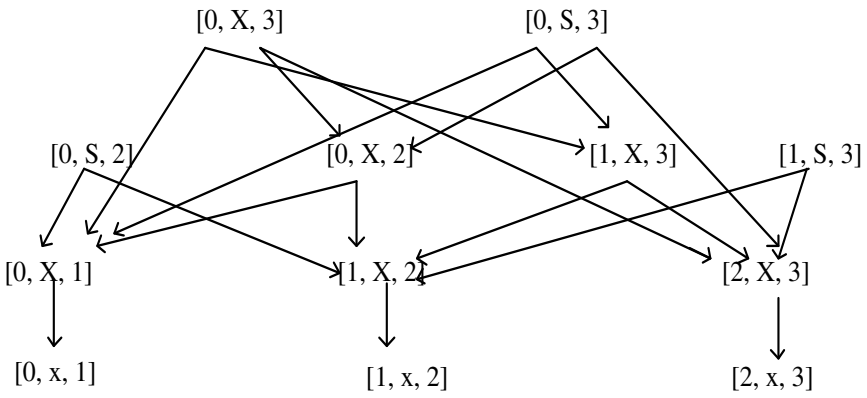


Figure 8.2: *dependency graph by CKY parsing*

Agenda is essentially a cache for items that are newly produced and remain to be processed. During each iteration, one item is pulled off from the agenda and added to the chart (on condition that no its duplicate in the chart). Take this item as a trigger, more items may be created by invoking the inference rules in the item-based description component and added into agenda. Repeat such iterations until agenda gets exhausted and all derivable items will be stored in the chart.

During this process, every time a new item is derived, it is inserted into the graph and its dependency relation is recorded as well. In our implementation, we sort the graph during the graph construction. Every time an OR-node is created, a partial sort is taken to keep the graph in a sorted state.

Try this example on the first approach, all possible items, such as $[0, S, 1]$ would be organized into the graph, whether the item is derivable or not. For the third approach, those items with the same length such as $[0, X, 1]$, $[1, X, 2]$ and $[2, X, 3]$ will be put into the same bucket and presents one node in the graph.

Dependency Graph and Shared Forests Structure In (Billot and Lang, 1989), the structure of shared forests in ambiguous parsing is discussed in detail. The structure of shared forests is precisely an AND-OR shared forest graph. One experimenting result in that paper has shown that *sophistication* may have a negative effect on the efficiency of all-path parsing. This result is verified again in our experiment. CKY and Earley algorithm are implemented respectively in our framework. It is also mentioned that one essential guideline to achieve better sharing

(and often also reduced computation time) is to try to *recognize every grammar rule in only one place of the generated chart parser code, even at the cost of increasing non-determinism.*

This guide essentially has the same idea with the third approach. Since in our system, the number of items and dependencies between items is the key factor in improving the time and space efficiency, we have to try to reduce the number of *buckets*, which equals to increase sharing of the structure in the graph on some level. However, this method will sacrifice the generality of system and take more programming efforts.

8.4.3 Interpreter Component

Once we have the bucketing, the parsing step is fairly simple. The basic algorithm appears in Figure 8.3. We simply loop over each item in each bucket. There are two types of buckets: looping buckets and nonlooping buckets.If the current bucket is a looping bucket, we compute the infinite sum needed to determine the bucket's values; in a working system, we substitute semiring-specific code for this section, as described in Section 3.3.3. If the bucket is not a looping bucket, we simply compute all of the possible instantiations that could contribute to the values of items in the bucket. Finally, we return the value of the goal item.

for *current* := first bucket **to** last bucket
  if *current* is a looping bucket
    /**replace with semiring-specific code*/
    for $x \in$ *current*
      $V[x, 0] = 0;$
    for $g := 1$ **to** ∞
      for each $x \in$ *current*, $a_1 \cdots a_k$ s.t. $\frac{a_1 \cdots a_k}{x}$
        if $a_i \in$ *current*
          $V[x, g] := V[x, g] \oplus \otimes_{i=1}^{k} V[a_i, g - 1]$
        else
          $V[x, g] := V[x, g] \oplus \otimes_{i=1}^{k} V[a_i]$
    for each $x \in$ *current*
      $V[x] := V[x, \infty];$
  else
    for each $x \in$ *current*, $a_1 \cdots a_k$ s.t. $\frac{a_1 \cdots a_k}{x}$
      $V[x] := V[x] \oplus \otimes_{i=1}^{k} V[a_i];$
return $V[goal];$

Figure 8.3: *Forward semiring parser interpreter.*

The reverse semiring parser interpreter is very similar to this forward one except that the buckets are traversed in reverse order.

The interpreter component executes this process by receiving a sorted item list from graph generation and sorting component and repeatedly invokes the corresponding item value computation and update component, which is essentially a specialized semiring, to compute and update the values for items in terms of the order.

8.4.4 Item Value Computation and Update Component

The definition of *item value* depends on a specific semiring. This component specializes the semirings. For example, in the Viterbi semiring, the item value is the largest probability of deriving the item; in Viterbi derivation semiring, item value is a pair of Viterbi value and a derivation list that can derive this item. So in this component, multiple semirings will be defined and a particular semiring will be invoked according to the particular task. In our implementation, a template semiring is defined first, including the definition of abstract multiplicative identity, additive identity, multiplication and addition. Every specific semiring is defined based on the template. Specifically, data type, multiplication, addition, multiplicative identity and additive identity are defined in every particular semiring. So the value computation and update gets easier when all these elements are specialized in the semiring.

8.5 Summary

Semiring parsing provides a general framework for all kinds of values computations involved in parsing. In this section, we described the basics and principles of semiring parsing, and implemented a preliminary semiring parsing system. This semiring parsing system construct a dependency graph based on deductive parsing and apply a variety of semirings to this structure in a unified way.

9 Efficiency Considerations in Semiring Parsing

Recall that in the implementation of basic semiring parsing system, we use a *dependency graph* to organize (and order) all items that are derived by the deductive parsing system. In this dependency graph, every item occurs only once and is shared by all those parse trees (if any) that contain it, which indicates that all possible parse trees are implicitly represented in the dependency graph without being enumerated one by one. Efficiency dealing with the creation of the dependency graph is not trivial. In this section, we will address it more extensively and formally.

One might notice that the dependency graph contains all derivable items. However, most of the items may not be contributive to the final result for a particular semiring. For example, in many applications, the goal is just to find the best parse, which means the semiring can be specialized into a Viterbi semiring. In this case, only a small subset of items is useful because only one parse is reported. If any useful information can be applied to guide the process of deduction, it will greatly reduce the size of derivable item set, and the dependency graph will get even smaller. If we consider (deductive) parsing as a search, then the integration of a good search strategy is of great importance. Such an approach is not only potentially faster it is usually more useful as well for the semiring system.

In the following, we will introduce and compare some current work on exploring the space efficiency and time efficiency in parsing. Most of search strategies currently used in parsing only focus on the particular problems such as Viterbi parsing on PCFGs.

9.1 Space Effciency : Parsing as Intersection

We already know that a parsing algorithm is one that, for any given string, can reconstruct the parse trees. Unfortunately it is often the case that sentences are highly ambiguous and the number of parses can be exponential in the size of the parsed sentence. Then enumerating all possible parses may be inconvenient and unnecessary for two reasons:

- The cost in time and space is proportional to the number of parses, while recognition or producing a single parse may be more efficient (e.g. polynomial in the size of the parsed sentence).
- Further processing has to be done separately on each of the parses, while

it is almost always the case that the corresponding structures have common subparts that could be shared, and then processed only once.

Hence, it is natural to produce as the result of a parsing algorithm a single structure that represents all possible parses with sharing of common subparts.

9.1.1 BPS Construction and Efficient Parsing Algorithms

In (Bar-Hillel et al., 1964), a well-known construction, referred to as BPS in that paper, showed that *the intersection of a context-free language $L(G)$ and a regular language $L(R)$ was again a context-free language*. A very useful property of the construction is that the resulting grammar $G_\cap$ is structured similarly to the original context-free grammar G, more precisely, $G_\cap$ encodes the set of all parse trees, formed according to G, of strings in $L(G) \cap L(R)$, and individual parse tree can be extracted in an effective manner. The time complexity of the construction and the size of the resulting grammar are both O(n^{p+1}), where n is the number of states of the corresponding finite state automaton of R, and p is the length of the right-hand side of the longest rule in G. Hence they are both cubic in the number of states of fsa of R when the grammar G is in so-called *2-form* or *binary form*.

By taking the finite automaton R to be of a special kind, accepting only a single string, this result anticipated many of the early tabular parsing algorithms, such as CKY algorithm, (Earley, 1970; Tomita, 1987; Schabes, 1991), and the algorithm for tabular simulation of pushdown automata from (Lang, 1974; Billot and Lang, 1989). All these dynamic programming CF parsing algorithms may be seen "optimizations" of the original BPS construction. The gain in performance comes from the fact that the raw BPS construction produces a grammar that contains a large number of useless rules and nonterminals and thus needs simplification, while the more recent constructions attempt to directly produce a grammar which is as clean as possible of useless components. For example, the well-known CKY construction avoids producing nonterminals and rules that are *nongenerating*. Most other constructions have top-down filters that eliminate some *inaccessible* nonterminals and rules, i.e., which would not be reachable from the initial symbol of the resulting grammar.

The construction of using grammars in binary form to get the cubic complexity applies to all these algorithms, whether explicitly (CKY algorithm) or implicitly ((Earley, 1970; Lang, 1974; Schabes, 1991)).

One final note is that BPS construction builds an intersection grammar $G_\cap$, however, without telling directly whether the string that was "parsed" is actually

recognized as being in the language $L(G)$. As noted by Aho (1968), this can be determined by checking if the language $L(G_\cap)$ is empty or not, which can be done in time linear in the size of $G_\cap$.

(Nederhof and Satta, 2003) extends BPS algorithm to a weighted case by computing the intersection of a probabilistic context-free language and a probabilistic finite automaton (PFA) provided the two probabilistic models are combined through multiplication. The result implies that for a PCFG G with probability function p_G and a PFA M with probability function p_M, we can find a PCFG $G_\cap$ with probability function $p_\cap$ such that:

$$p_\cap(w) = \frac{1}{C} \cdot p_G(w) \cdot p_M(w)$$

for each string w, where C is a constant determined by G and M.

9.1.2 Shared Forests

BPS construction brings out the fundamental concepts that underlie some well-known efficient algorithms which are usually called *chart or dynamic programming parsers*. The key is that the chart parsing is essentially equivalent to a simple construction of the intersection of the language (represented by its grammar) with a regular set containing only the input sentence to be parsed (represented by a finite state machine). The resulting grammar for that intersection is precisely what is usually called *a shared forest* (Billot and Lang, 1989): it represents simultaneously all the parses of a syntactically ambiguous sentence.

In Section 8.4.5 we have talked a little bit about shared forest structure. In fact dependency graph is basically a shared forest structure. In general case, a shared forest structure is represented by an AND-OR graph with the nodes being assigned different semantics in different applications, and to our knowledge, this representation is the simplest and most tractable theoretical formalization. In the case of context-free languages, the shared forest structure can be produced in cubic time and space complexity by some dynamic programming parsing algorithms, and any specific parse can be extracted linearly in the size of the extracted parse tree.

As we mentioned earlier, the shape of shared forest is tightly related to the parsing schema used. Typically, the simpler the parsing schema is, the best sharing can be achieved (Billot and Lang, 1989).

(Lang, 1994) analyzes the idea of representing the shared forest of an ambiguous sentence as a ϕ-grammar (where ϕ is the type of formalism considered) and

shows its applicability to a variety of known formalisms. This idea links dynamic programming parsing and intersection with regular sets, thus giving strong hints to the design of dynamic programming parser, notably with regard to complexity and sharing control, even though optimizing these parsers remains a substantial task. It also shows that ill-formed input processing techniques, based mostly on finite-state models, can be generalized in the same way.

However, shared forests must not only be constructed easily enough, but they must also be structurally meaningful. Furthermore they must be usable with sufficient ease, preferably in linear time to generate individual trees from the forest.

These criteria show that "mildly context-sensitive formalisms" such as TAGs (tree-adjoining grammars), LCFG (linear context-free grammars) etc. can be used as backbone syntactic formalisms with regard to several issues of linguistic processing. But too much power may quickly lead to intractability of these issues because of the complexity of the shared forests.

9.2 Time Efficiency : Integration of Search Strategies in Parsing

A shared forest structure provides a overall framework of organizing the intersection (such as the intersection of regular languages and context-free languages) in an efficient way. If take parsing as search, a good search strategy plays an important role, especially for a particular task, such as finding the best parse. Currently, a lot of work is focused on finding the most probable parse which is often reduced to "the shortest distance problem". And some efficient algorithms have been explored by taking into account the probabilities in parsing.

9.2.1 Non-heuristic Shortest-Distance Algorithms in Parsing

Finding the most probable parse can be formalized into solving the minimization problem or reduced to "searching the shortest distance problem". Most solutions to this problem is based on the basic "single-source shortest path algorithm".

"Single-source shortest path algorithm", called Dijkstra's algorithm as well, is well-known for solving the problem of finding the shortest paths from a single source vertex to all other vertices in a weighted, directed graph, when the weight on each arc is nonnegative. It basically takes a breath-first search strategy and maintains a priority queue in the implementation. This algorithm can be found in most literatures of data structure and algorithms. We omit the details here.

Knuth's Generalized Algorithm (Knuth, 1977) presented an algorithm which generalized Dijkstra's in essentially the same way that tree structures generalize linear lists, or that context-free grammars generalize regular languages.

This algorithm is used to compute the *smallest* values corresponding to the languages, namely, $m(Y) = \min\{\text{val}(\alpha) \mid \alpha \in L(Y),\ Y \in N\}$ for a grammar $G = (\Sigma, N, P, S)$ and $L(Y) = \{\gamma \mid Y \Rightarrow^* \gamma\}$. This algorithm guarantees to find the optimal solution, however, it requires the "weight" functions be monotonically nondecreasing (called *superior* in the paper), therefore a longer "path" can never be extended to become a shorter "path". The running time of this algorithm is bounded by a constant time $m\log n + t$, where there are m productions and n nonterminals, and the total length of all productions is t. Even though this algorithm is not immediately related to the computation of most likely parse, all the current methods are basically based on its idea. See (Knuth, 1977) for the most details and proof for the correctness of the algorithm.

Knuth's Algorithm in Weighted Deductive Parsing (Nederhof, 2003) discussed a modular design for weighted deductive parsing by distinguishing between a weighted deduction system, on the one hand, which pertains to the choice of grammatical formalism and parsing strategy, and the algorithm that finds the derivation with the lowest weight, on the other. The latter is Knuth's generalization of Dijkstra's algorithm for the shortest-path problem on grammars.

Same as the condition for Knuth's algorithm, the weight functions f have to be *superior* as well, which means that they are monotone nondecreasing in each variable and the $f(x_1, \cdots, x_m) \geq \max(x_1, \cdots, x_m)$ for all possible values of $x_1, \cdots, x_m$.

Knuth's Algorithm and Viterbi Algorithm (Nederhof, 2003) also addressed the efficiency of Knuth's algorithm for weighted deductive parsing, relative to the more commonly used Viterbi algorithm. We have mentioned before that there are some constraints for the application of Knuth's algorithm. However, this algorithm is applicable on a weighted deduction system if a simple partial order on items exists such that the antecedents of an inference rule are always strictly smaller than the consequent. When this is the case, we may treat items from smaller to large to compute their lowest weights. Therefore, no constraints (such as monotonically nonincreasing) are posed on the weights functions any more. Viterbi algorithm is one of those that operate according to this principle. The partial order is based on the linear order given by a string of input symbols. A

special situation arises when a deduction system is such that inference rules allow cyclic dependencies within certain subsets of items, but dependencies between these subsets represent a partial order. One may then combine these two algorithms: Knuth's algorithm is used within each subset and Viterbi algorithm is used to relate items in distinct subsets (Bouloutas et al., 1991).

In cases in which both Knuth's algorithm and Viterbi algorithm are applicable, the main difference between the two is that Knuth's algorithm may halt as soon as the lowest weight for a goal item is found, and no items with larger weights than that goal need to be treated, whereas Viterbi algorithm treats *all* derivable items. This suggests that Knuth's algorithm may be more efficient than Viterbi's. The worst case time complexity of Knuth's algorithm, however, involves an additional factor due to the maintenance of the priority queue. This factor is $O(\log(\|c(G,w)\|))$, where $\|c(G,w)\|$ is the number of nonterminals in context-free grammar G with input w (denoted $c(G,w)$), which is the upper bound on the number of elements on the priority queue at any given time. Furthermore, there are observations suggesting that apparent advantage of Knuth's algorithm does not necessarily lead to significantly lower time costs in practice. This is not hard to understand in the scenario that in deductive systems with items associated with spans, Knuth's algorithm treats most items with smaller spans before any items with a larger span is treated, and since goal items typically have the maximal span, covering the complete input, there are fewer derivable items at all that are not treated before any goal item is found.

Knuth's Algorithm Using Hypergraph (Klein and D.Manning, 2001) presented a view of taking parsing as directed *hypergraph* analysis, which offered an *indivisible* specification for both symbolic parsers and probabilistic parsers. Knuth's algorithm is applied on the hypergraph with a small collection of parsing strategies for weighted context-free grammars. They restated the deep connection between parsing and hypergraph. And with this connection, the existing graph algorithms could be exploited in probabilistic parsing.

In their framework, parse existence corresponds to the reachability in a certain hypergraph and the Viterbi parsing basically corresponds to find the shortest path in the hypergraph.

This framework is similar to the semiring framework to some extent. In particular, the hypergraph is precisely a shared forest or a dependency graph with nodes having the same form as items [ref. Section 7, 8]. However, it prefers an indivisible way to combine the specification of parsing strategies and the value

computation, which limits its generalization across parsing tasks.

9.2.2 Heuristic Search for Shortest-Distance in Parsing

PCFG parsing algorithms with worst case cubic-time bounds are well-known. However, when dealing with wide-coverage grammars and long sentences, even cubic algorithms can be far too expensive in practice. By taking parsing as a search process, the classic heuristic search strategies can be extended to the probabilistic parsing and may speed up the parsing speed dramatically. In the following, we will review some work of using various heuristic information in Viterbi parsing. And we assume the readers are familiar with the basic heuristic searching methods.

Best-first Search Typically, in chart parsing one maintains an agenda of items remaining to be processed, one of which is processed during each iteration. As each item is pulled off the agenda, it is added to the chart (unless it was already there) and used to extend and create additional items. "Exhaustive" chart parsing removes items from the agenda iteratively until nothing remains.

Some alternatives differing from a classic chart parser are that agenda items are processed according to a *priority*. Knuth's algorithm gives this priority without using any heuristic information and it guarantees to give the optimal solution. Another commonly used alternative, which is more efficient as well, is called *best-first parser* (Charniak et al., 1998). In best-first parsing, one builds a *figure of merit* (FOM) over items in the agenda, and uses the FOM to decide the order in which agenda items should be processed.

This approach also dramatically reduces the work done during parsing, though it gives no guarantee that the first parse returned is the actual Viterbi parse.

Beam Search Strategy (Ratnaparkhi, 1999) uses a *beam-search* strategy, in which only the best n (in (Ratnaparkhi, 1999), $n = 20$) parses are tracked at any moment. The advantage of this method is that one can incorporate beam search strategies into existing parsing algorithms without any significant additional processing cost. And parsing time is linear and can be made arbitrarily fast by reducing n. As a greedy strategy, however, it has a major drawback: the optimal solution (typically, Viterbi parse) can be pruned from the beam because, while it is globally optimal, it may not be locally optimal at every parse stage.

A^* **Search** (Klein and Manning, 2003) presented an extension of the classical A^* search procedure. Let $\beta(e, s)$ denote the log-probability of a best inside parse of e where e is an edge and s is the input (its inside score). The parse maintains estimate $b(e, s)$ of $\beta(e, s)$ which increases over time and always represent the score of the best parses of their edges e discovered so far. Obviously, using $b(e, s)$ to prioritize edges is basically the strategy taken in Knuth's algorithm.

Introduce $a(e, s)$ to estimate $\alpha(e, s)$ (the score of outside parse of e), and add it to $b(e, s)$ to focus exploration on regions of the graph which appear to have good *good* cost.

To satisfy the requirement of heuristic function of A^*, use $b + a$ as the edge priority provided a is an admissible, monotonic estimation of α. So the estimation of α is the crux of this extended A^* algorithm. In (Klein and Manning, 2003), assuming G is a binarized grammar, two estimation methods are explained. One way to construct an admissible estimate is to summarize the context in some way, and to find the score of the best parse of *any* context that fits that summary. This method is called *context summary estimates*. For the other one called *grammar projection estimates*, the exact context is used, but the original grammar G is projected to some G' which is so much simpler that it is feasible to first exhaustively parse with G' and then use the result to guide the search in the full grammar G. Even though the resulting grammar will not generally be a proper PCFG, all that matters is that every tree in G projects under this projection to a tree in G' with the same or higher probability. Therefore, $\alpha_G(e, s) \leq \alpha_{G'}(e, s)$.

In summary, the search strategies discussed here are essentially the extensions of the classical search strategies from linear structures/regular grammar to tree structures /context-free grammar. Specifically, the "Viterbi inside value" and "Viterbi outside value" of some particular constituents become of special interest when taking into account the good choice of cost estimates.

The search strategies introduced in this section basically work on the Viterbi parsing which is the most common parsing task. Trying to combine search strategies with particular semiring parsers is one of our future work.

10 Structured Probabilistic Parsing Models

In the previous parts, we were mainly working on PCFGs. As we mentioned before, PCFG is just a simple probabilistic model in which *features* are narrowly defined with *productions rules or local trees* and we organize all such features

into a dependency graph structure. In this section, we will discuss more structured probabilistic models beyond PCFGs and dependency graph. We will find out how surprisingly similar for all these formalisms.

10.1 Feature Forests

In Section 9.1.2, we discussed the *shared forest structure* in detail. Recall that a shared forest, which is basically an AND-OR directed acyclic graph (DAGs), is a packed representation to represent all possible parses without enumerating them one by one. Dependency graph as one form of shared forest structure is applied to the current semiring parsing system. Notice that the construction of "shared forest structure" we had is based on the probabilistic context-free grammars (PCFGs) which is a simplistic probabilistic model for syntactic structures. In a PCFG model, an independence assumption is made that all dependencies are *local* or *context-free*, and *features* only have "context free" branching structures (i.e., production rules or subtrees) to simplify the problem.

However, syntactic structures may exhibit non-local constraints and we have no reason to only use local tree as features. For example, We need a more general definition for "features" with less constraints and we need a more complete structure to organize the features. And we need a more flexible model than PCFGs to deal with the features as well.

(Miyao and Tsujii, 2002) presented a more complete structure of features, which is called "feature forests", and proposed an algorithm by dynamic programming similar to the classic *inside-outside algorithm* to do parameter estimation without unpacking the feature forest. This algorithm is proposed for *maximum entropy modeling* (Berger et al., 1996; Ratnaparkhi, 1997) since it allows incorporating various overlapping features of a complete structure without independence assumptions, and it can be applied to a probabilistic model of events that are difficult to divide into independent sub-events[7].

10.1.1 The Structure of a Feature Forest

Similar to shared forests structure, a feature forest is a packed structure that is used for enumerating all possible structures of target events which are all possible

[7]When modeling a probabilistic distribution of event $e = (h, t)$, where h is a *history event* and t a *target event*, e is represented by a bundle of *feature functions* (or *features* for short) $f_i(e)$. A *feature function* represents the existence of a certain characteristic in event e, and a set of *activated* features, i.e., $f_i(e) \neq 0$, is considered to be an abstracted representation of an event.

parses for parsing task. More generally, a node in a feature forest can represent any linguistic entity, including a fragment of a syntactic/semantic structure and other sentence-level information. And the substructures of a feature forest have no independence assumptions as shared forests do, which gives this structure more flexibility and generality. In this packed representation, a set of *conjunctive nodes* and *disjunctive nodes* are defined respectively. And feature functions are defined over conjunctive nodes. Thus a feature forest represents a set of trees of features, which are associated to conjunctive nodes. A feature forest is essentially a directed acyclic AND-OR graph.

Definition A feature forest Φ is a 5-tuple $(C, D, r, \gamma, \delta)$, where

- C is a set of conjunctive nodes
- D is a set of disjunctive nodes
- r is the root node: $r \in C$
- $\gamma : D \mapsto 2^C$ is a conjunctive daughter function
- $\delta : C \mapsto 2^D$ is a disjunctive daughter function

Figure 10.1 shows an example feature forest. In this forest, $C = \{c_1, c_2, c_3, c_4, c_5\}$

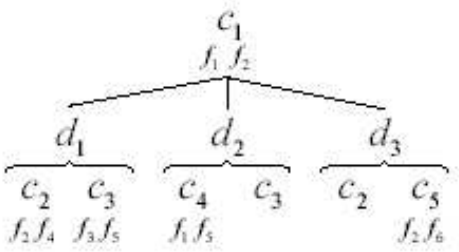


Figure 10.1: *A feature forest*

is a set of conjunctive nodes, $D = \{d_1, d_2, d_3\}$ is a set of disjunctive nodes, $r = c_1$ is the root node, $\gamma(d_1) = \{c_2, c_3\}, \gamma(d_2) = \{c_4, c_3\}, \gamma(d_3) = \{c_2, c_5\}$, $\delta(c_1) = \{d_1, d_2, d_3\}, \cdots$. Each disjunctive node has alternative nodes, which are conjunctive nodes. Each conjunctive node has activated features, i.e., $f_i(c) \neq 0$; it also has disjunctive nodes as daughters.

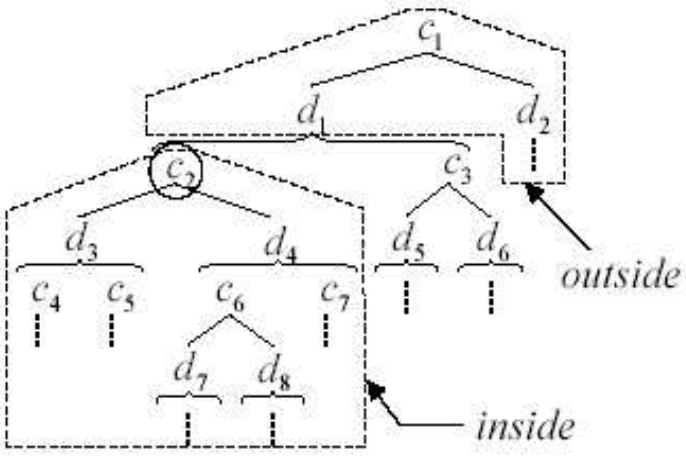


Figure 10.2: *Inside/outside at node c_2 in a feature forest*

Since Figure 10.1 resembles a shared (parse) forest of a PCFG, it might be asserted that a feature forest can represent only immediately dominance relations in each node, as in a PCFG, resulting in only a slight, trivial extension of the PCFG. However, as we have pointed out, the nodes in a feature forest can be associated with the most general semantics. It is a generalized representation of an ambiguous structure, rather than limited to parse structures, and each node in a feature forest does not need to correspond to a node in a shared (parse) forest.

10.1.2 Inside-Outside Algorithm for Parameter Estimations in Feature Forests

Parameter estimation is a fundamental task in statistical learning. For example, for statistical parsing, maximum entropy model p_M gives a conditional probability of event $e = (h, t)$, where $\tau(h)$ is a set of targets (parses) observable with history (input sentence) h:

$$p_M(t|h) = \frac{1}{Z_h} \prod_i \alpha_i^{f_i(h,t)}$$

$$\text{where } Z_h = \sum_{t' \in \tau(h)} \prod_i \alpha_i^{f_i(h,t')}$$

A feature function is an indicator for a certain characteristic of an event, and *model parameter* α_i corresponds to f_i is its weight. A maximum entropy model assigns a probability to an event by multiplying weights α_i when the corresponding features are activated, i.e., $f_i(e) \neq 0$. A maximum entropy model yields a

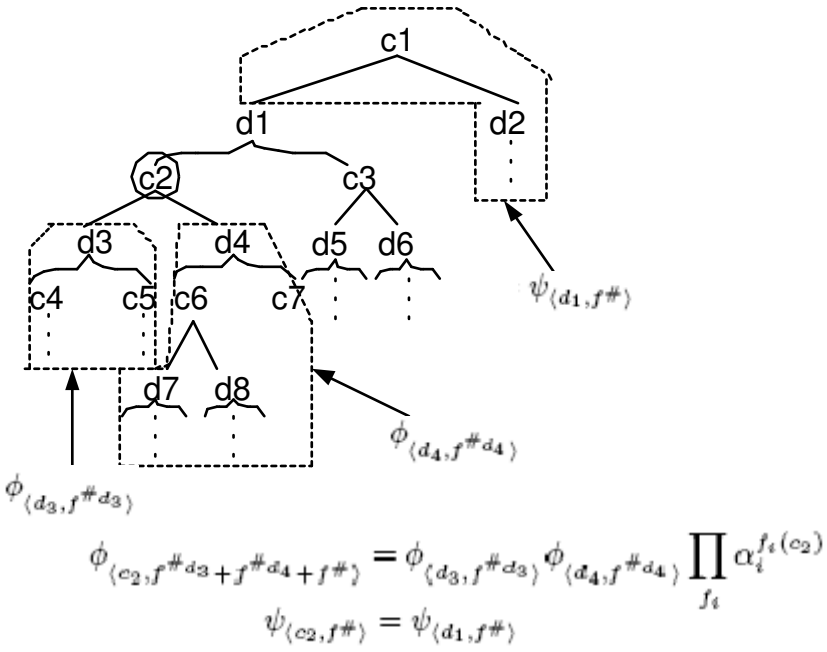


Figure 10.3: *Incremental computations of inside α-products ϕ and outside α-product ψ at conjunctive node c_2*

probability distribution that maximizes the likelihood of the training data given a set of feature functions.

In this section, we just use some figures to illustrate the basic idea and definitions for parameter estimations on feature forests. Once we understand the definitions of the *inside/outside α-products* over conjunctive and disjunctive nodes, the proposed algorithm essentially has the same idea as the classic inside-outside algorithm for PCFGs, which can be found in Appendix A.

Figure 10.2 illustrates the *inside/outside* concepts for a conjunctive node, which is similar to that of a PCFG. For node c_2, *inside* denotes a set of partial trees (conjunctive nodes) derived from node c_2, and *outside* denotes the complement of the inside trees.

The key point of the algorithm is to compute inside α-products ϕ and outside α-products ψ for each node in C, not for all unpacked structures. The α-products are computed by dynamic programming. Figure 10.3 shows the process of com-

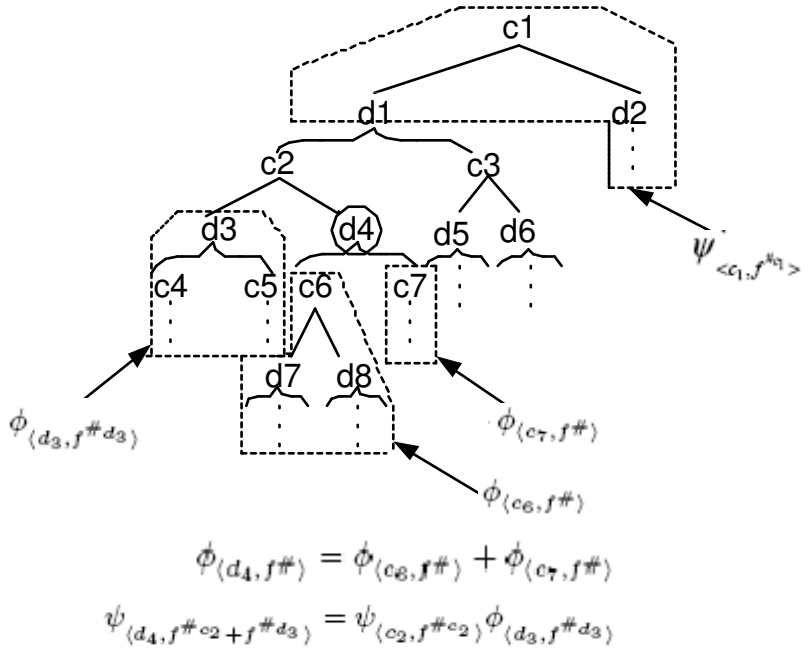


Figure 10.4: *Incremental computations of inside α-products ϕ and outside α-product ψ at disjunctive node d_4*

puting the inside α-product and the outside α-products at conjunctive node c_2. Figure 10.4 shows the computation process of computing α-inside and outside products for disjunctive node d_4. And $f^{\#}$ is introduced for the algorithm optimization by factoring some term.

It has to be pointed out the order of traversing nodes is important for incremental computation. The computation for the daughter nodes and mother nodes must be finished before computing the inside and outside α-product, respectively. The inside α-product is computed in a bottom-up fashion and outside top-down. This constraint is quite understandable and can be easily solved using a topological sort. This computation fashion is exactly the same as the one in the dependency graph. The complexity of the algorithm is $O(|C| + |D|)|\varepsilon||F|)$, where $|C|, |D|$, $|\varepsilon|$, $|F|$ are the number of conjunctive nodes, disjunctive nodes, events and the average number of activated features respectively.

One problem of this algorithm is it is somewhat limited by the size of the forest and due to the huge number of features, it may be more costly than the

conventional ones. Also it is subtle to organize the features into the forest (Miyao, 2003) and for higher efficiency, some best-first/beam search can be posed on the algorithm.

10.1.3 Summary

Feature forests provide a more general and complete representation for an ambiguous structure. It encodes all the possible structures for the target event in a packed structure with no independence assumptions in the maximum entropy modeling. Compared with the dependency graph for PCFGs, it enables a more general and flexible model for various NLP tasks because it allows incorporating various overlapping features of a complete structure. Based on the definition of inside/outside α-products, the classic inside-outside algorithm can be applied to this structure to do parameter estimation. A similar problem occurs when the size of the forest is too large to be tractable. How to formalize this more powerful representation and parameter estimation into semiring framework is one of our future work.

10.2 MK Packed Representations and Graphical Models

In (Geman and Johnson, 2002), a graph-based dynamic programming algorithm was described for calculating the statistics from the packed unification-based grammars (UBGs) parse representations of Maxwell and Kaplan (*MK packed representations*). The key observation is that by using MK packed representations, the required statistics can be rewritten as either *the max or the sum of a product of functions*. This is exactly the kind of problem which can be solved by dynamic programming over *graphical models*.

10.2.1 MK packed representations

A parse generated by a unification grammar is a finite subset of a set $\mathcal{F}$ features. Features are parse fragments, e.g., chart edges or arcs from attribute-value structures, out of which the packed representations are constructed. We can just take features as the atomic entities manipulated by a dynamic programming parsing algorithm. A grammar defines a set Ω of well-formed or grammatical parses. Each parse $\omega \in \Omega$ is associated with a string of words $Y(\omega)$ called its *yield*.

If y is a string, then let $\Omega(y) = \{\omega \in \Omega \mid Y(\omega) = y\}$ and $\mathcal{F}(y) = \bigcup_{\omega \in \Omega(y)} \{f \in \omega\}$. That is, $\Omega(y)$ is the set of parses of a string y and $\mathcal{F}(y)$ is the set of features

appearing in the parses of y. In the grammars of interest $\Omega(y)$ and hence also $\mathcal{F}(y)$ are finite.

MK packed representations are similar to shared forests or feature forests in the sense that it provides a more compact representation of a set of parses of a sentence.

Definition A *packed representation* of a finite set of parses is a 4-tuple $R = (\mathcal{F}', X, N, \alpha)$, where:

- $\mathcal{F}' \supseteq \mathcal{F}(y)$ is a finite set of features
- X is a finite vector of *variables*, where each variable X_l ranges over the finite set $\mathcal{X}_l$
- N is a finite set of conditions on X called *no-goods*[8]
- α is a function that maps each feature $f \in \mathcal{F}'$ to a condition α_f on X.

Packed feature representations are defined in terms of conditions on the values assigned to a vector of variables X which is called *context variables* in MK representations. These variables have no direct linguistic interpretations. More formally, a vector of values x[9] *satisfies the no-goods* N iff $N(x) = 1$, where $N(x) = \prod_{\eta \in N} \eta(x)$. Each x that satisfies the no-goods *uniquely identifies* a parse $\omega(x) = \{f \in \mathcal{F}' \mid \alpha_f(x) = 1\}$, i.e., ω is the set of features whose conditions are uniquely satisfied by x. Finally, a packed representation R represents the set of parses $\Omega(R)$ that identified by values that satisfy the no-goods, i.e., $\Omega(R) = \{\omega(x) \mid x \in \mathcal{X}, N(x) = 1\}$

(Maxwell III and Kaplan, 1981) describes a parsing algorithm for unification-based grammars that takes as input string y and returns a packed representation R such that $\Omega(R) = \Omega(y)$, i.e., R represents the set of parses of the string y. Here we just focus on the basic idea of packed representation and the calculations based on graphical models with more details omitted.

10.2.2 Graphical Model Calculations

Based on the representations discussed above, for a given sentence w, we can let $W'(x) = W(y)$ where y is the parse identified by x and W is a product of

[8]The name "no-good" comes from the TMS (Truth Maintenance System) literature, and was used by Maxwell and Kaplan. However, here the no-goods actually identify the *good* assignments.

[9]we use X to denote variables and x denote values.

weight functions of y. So those computations over parse y can be taken over to context variables X by this transformation. In statistical parsing, the quantities involving the maximization or summation over a product of functions, each of which depends only on the values of a subset of the values X, are commonly used to find the target parse or take the parameter estimations. There are dynamic programming algorithms for calculating all of these quantities, which are now relatively standard; the most well-known approach involves *junction trees* (Pearl 1988). These graph algorithms are rather involved in this case.

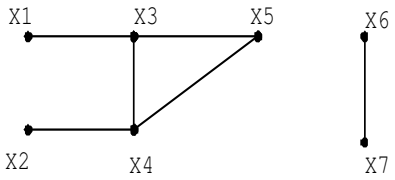
We can represent the Viterbi function and Inside function for input sentence w as follows respectively:

$$\text{Vit}(w) = \max_x W'(x) = \max_x \prod_{A \in \mathcal{A}} A(x)$$

$$\text{Inside}(w) = \sum_x W'(x) = \sum_x \prod_{A \in \mathcal{A}} A(x)$$

where $\mathcal{A}$ is a set of functions such that the quantities we need to calculate have the general forms as above.

According to the definitions above, we organize the "dependency graph" (there is a partial order posed on the nodes, even though not shown in the graph) $G_{\mathcal{A}}$ for $\mathcal{A}$. Context variables X are vertices of $G_{\mathcal{A}}$ and (X_i, X_j) is an edge of $G_{\mathcal{A}}$ iff both are arguments of some $A \in \mathcal{A}$. Look at the following example for computing the inside value based on the context variable (Boolean variables):



$$
\begin{aligned}
A(X) &= a(X_1, X_3)b(X_2, X_4)c(X_3, X_4, X_5)d(X_4, X_5)e(X_6, X_7) \\
Z &= \sum_x W'(x) = \sum_x \prod_{A \in \mathcal{A}} A(x) \\
&= \Sigma_x a(x_1, x_3)b(x_2, x_4)c(x_3, x_4, x_5)d(x_4, x_5)e(x_6, x_7) \\
Z_1(x_3) &= \Sigma_{x_1} a(x_1, x_3) \\
Z_2(x_4) &= \Sigma_{x_2} b(x_2, x_4) \\
Z_3(x_4, x_5) &= \Sigma_{x_3} c(x_3, x_4, x_5) Z_1(x_3) \\
Z_4(x_5) &= \Sigma_{x_4} d(x_4, x_5) Z_2(x_4) Z_3(x_4, x_5) \\
Z_5 &= \Sigma_{x_5} Z_4(x_5) \\
Z_6(x_7) &= \Sigma_{x_6} e(x_6, x_7) \\
Z_7 &= \Sigma_{x_7} Z_6(x_7) \\
Z &= Z_5 Z_7 \\
&= (\Sigma_{x_5} Z_4(x_5))(\Sigma_{x_7} Z_6(x_7))
\end{aligned}
$$

The computational complexity of the algorithm is polynomial in m, which is the maximum number of variables in the dynamic programming functions, and m depends on the ordering of variables (and G). Finding the variable ordering that minimizes m is NP-complete (Geman and Johnson, 2002), but there are good heuristics.

10.2.3 Summary

The calculations (mainly involve maximum and sum operations) based on graphical models are analogues of the dependency graph for PCFGs we discussed previously; specifically all these computations are based on some packed representations, and the nodes in the representations are posed with a certain order. The dynamic programming takes effect by taking advantage of this order. With MK representation, the computations over parses y are taken over to context variables X, which is different from the previous implementations. As for how this transformation is taken, refer to (Maxwell III and Kaplan, 1981).

The statistical techniques described here can be extended to non-linguistic applications of TMSs as well. One problem is the features here must be local to parse fragments, which means we cannot define feature functions arbitrarily and for a parse y and some feature function f, the equation $f(y) = \Sigma_{\xi \in y} f(\xi)$ has to hold. Currently, an empirical evaluation of the algorithms described here are still needed. And it seems reasonable to expect that if the linguistic dependencies

in a sentence typically factorize into largely non-interacting *cliques* [10] then the dynamic programming methods may offer dramatic computational savings compared to current methods that enumerate all possible parses. Another interesting work would be to do some comparison between this dynamic programming algorithms and the one described by (Miyao and Tsujii, 2002).

Another similar work can be found in (M.Aji and J.McEliece, 2000). For the node denoted by some context variable X, the product from its preceding nodes whose values have been computed is essentially the *messages* it receives from all its neighbor nodes. And the function defined at this node is correspondingly a local *kernel* in the *MPF(marginalize a product function) problem*.

10.3 Case-Factor Diagrams

In (McAllester et al., 2004), CFD (case-factor diagrams), a more generalized structured probabilistic modeling formalism, is introduced. This modeling formalism can handle both Markov random fields (MRFs) of bounded tree width and probabilistic context-free grammars (PCFGs). In particular, the well-known inference algorithms (Inside-Outside algorithm) for PCFGs which runs in cubic time in the length of the word string x can be nicely applied based on this formalism.

10.3.1 Motivations and Linear Boolean Model (LBM)

A *linear Boolean model* (LBM) is a general class of structured probabilistic models with Boolean variables. A *linear Boolean model* (LBM) is a triple (V, F, Ψ) where V is a set of boolean variables, F is a set of *feasible configurations*, each of which is a truth assignment to V; and $\Psi : V \rightarrow R$ is an *weight function* (energy function). The weight for a complete variable assignment is then the sum of weights for those variables in the assignment that are true. So the weight associated with a truth assignment can be written as a linear function of the bits in the assignment.

In the framework of LBM, the main problem involved then is how to encode compactly the set of all the possible assignments to the variables in an LBM in a single formalism handling both MRFs of bounded tree width and PCFGs, especially for the sparse truth assignments – truth assignments where most of the Boolean variables are false since sparseness is important for representing PCFGs.

[10] A *clique* of a graph is its maximal complete subgraph. A complete graph is a graph in which each pair of graph vertices is connected by an edge.

10.3.2 Case-Factor Diagrams (CFDs)

CFD is introduced right for this purpose. It represents the feasible set by a search tree over the set of possible truth assignments. The search tree cases on the values of individual variable and factors the feasible sets into a product of independent feasible sets when possible. Formally, this case-factor search tree can be represented by an expression.

Definition A *case-factor diagram* (CFD) D is an expression generated by the following grammar where x is a Boolean variable; a case expression $case(x, D_1, D_2)$ must satisfy the constraint that x does not appear in D_1 or D_2; and a factor expression $factor(D_1, D_2)$ must satisfy the constraint that no variable occurs in both D_1 and D_2.

$$D ::= case(x, D_1, D_2) \mid factor(D_1, D_2) \mid unit \mid empty$$

We denote by $V(D)$ the set of variables occurring in D.

To define the meaning of CFDs, it is convenient to see all CFD variables as members of a common countably infinite set of variables V. The interpretation of $F(D)$ of a CFD D is then a finite set of finite support assignments to V. We use $\bar{0}$ for the totally false assignment (the zero vector). $F(D)$ is defined as follows:

$$\begin{aligned} F(unit) &= \{\bar{0}\} \\ F(empty) &= \emptyset \\ F(case(x, D_1, D_2)) &= F(D_1)[x := 1] \cup F(D_2) \\ F(factor(D_1, D_2)) &= F(D_1) \vee F(D_2) \end{aligned}$$

10.3.3 Parse Distributions as LBMs

For a given string x we have a probability distribution on parse tree y with yield$(y) = x$ defined as follows:

$$\begin{aligned} P(y|x) &= \frac{1}{Z(x)} e^{-\Psi(y)} \\ Z(x) &= \sum_{y: yield(y)=x} e^{-\Psi(y)} \end{aligned}$$

where $\Psi(y)$ is the total weight of the parse tree y.

To construct an LBM representation of $P(y|x)$ we first define a set of Boolean variables. Let n be the is the length of x. Suppose each production rule is in the form of CNF.

- Define a *phrase variable* "$X_{i,j}$" for each nonterminal X in the grammar and $1 \leq i < j \leq n+1$. This phrase variable represents the statement that the parse contains a phrase with nonterminal X spanning the string from i to $j-1$ inclusive.

- Define a *branch variable* "$X_{i,k} \rightarrow Y_{i,j} Z_{j,k}$" for each production rule $X \rightarrow YZ$ in the grammar and $1 \leq i < j < k \leq n+1$. A branch variable represents the statement that the parse contains a node labeled with the given production rule where the left child of the node spans the string from i to j_1 and the right child spans j to k_1.

- Define a *terminal variable* "$X_{i,i+1} \rightarrow a$" for each terminal production rule $X \rightarrow a$ and position i in the input string. A terminal variable represents the statement that the parse tree produces terminal symbol a form nonterminal X at position i.

Take V to be the set of all such phrases, branch, and terminal variables. Each parse tree determines a truth assignment to the variables in V and let F to be the set of assignments corresponding to parse trees.

Finally, we have to assign the weight for each Boolean variable. The weight function Ψ, which is called energy function here, is given by the following equations:

$$\begin{aligned} \Psi(\text{“}X_{i,j}\text{”}) &= 0 \\ \Psi(\text{“}X_{i,k} \rightarrow Y_{i,j} Z_{j,k}\text{”}) &= \Psi(X \rightarrow YZ) \\ \Psi(\text{“}X_{i,i+1} \rightarrow a\text{”}) &= \Psi(X \rightarrow a) \end{aligned}$$

The definitions above remind us of the forms of items in the previous item-based descriptions. There is a one-to-one correspondence between the Boolean variables here and the items there.

10.3.4 CFDs for Parsing

According to the formalization of CFD, here we construct a CFD for the feasible set of the LBM defined above for a grammar G. We define the CFD $D(\text{“}X_{i,k}\text{”})$

such that the assignment in $F(D(\text{“}X_{i,k}\text{”}))$ are in one-to-one correspondence with the parse trees of the span from i to $k-1$ with root nonterminal X. The CFD representing the full feasible set of parses is $D(\text{“}S_{1,n+1}\text{”})$. First we define $D(\text{“}X_{i,k}\text{”})$ as follows where $B(\text{“}X_{i,k}\text{”})$ represents the consequences of making “$X_{i,k}$” true.

$$D(\text{“}X_{i,k}\text{”}) = case(\text{“}X_{i,k}\text{”}, B(\text{“}X_{i,k}\text{”}), empty)$$

We use the notation $case(\langle z_1, D_1\rangle, \langle z_2, D_2\rangle, \cdots, \langle z_m, D_m\rangle)$ as an abbreviation for $case(z_1, D_1, case(\langle z_2, D_2\rangle, \cdots, \langle z_n, D_n\rangle))$ where $case(\langle z, D\rangle)$ is $case(z, D, empty)$. Based on the notation, for $k > j+1$ we define the consequences $B(\text{“}X_{i,k}\text{”})$ as follows:

$$B(\text{“}X_{i,k}\text{”}) = case(\langle b_1, B(b_1)\rangle, \cdots, \langle b_n, B(b_n)\rangle)$$

where the variables b_p are all possible branch variables of the form “$X_{i,k} \rightarrow Y_{i,j}Z_{j,k}$”, and $B(\text{“}X_{i,k}\text{”}) = factor(D(\text{“}Y_{i,j}\text{”}), D(\text{“}Z_{j,k}\text{”}))$.

Finally, if a_i is the ith input symbol, we have

$$B(\text{“}X_{i,i+1}\text{”}) = \begin{cases} case(\text{“}X_{i,i+1} \rightarrow a_i, unit, empty\text{”}) & \text{if } X \rightarrow a_i \in G; \\ \text{empty} & \text{otherwise.} \end{cases}$$

This construction has the property that $|D(\text{“}S_{1,n+1}\text{”})|$ is $O(|G|n^3)$ where $|G|$ is the number of production rules in the grammar. From the definitions above, we see that CFD expressions can be represented as a *diagram*, which are a DAG with each node for each distinct subexpression, and edges from the node for an expression to the nodes for its immediate subexpressions. That is, common subexpressions are represented uniquely. In the example in Section 4.3.2, and compare the resulting diagram and the dependency graph of Figure 8.2, we see these two representations have surprising similarities.

10.3.5 Inference on CFD Models

A CFD model $\langle D, \Psi\rangle$ is an LBM whose feasible set is defined by a CFD D and whose energy function Ψ assigns weights to the variables of D. In this part, we will present the main inference algorithms on CFDs.

The Inside Algorithm We use $Z(D, \Psi)$ to denote the *inside value* for D. It turns out that $Z(D, \Psi)$ can be computed by recursive descent on subexpressions

of D using the following equations:

$$\begin{aligned} Z(case(x, D_1, D_2), \Psi) &= e^{-\Psi(x)} Z(D_1, \Psi) + Z(D_2, \Psi) \\ Z(factor(D_1, D_2), \Psi) &= Z(D_1, \Psi) Z(D_2, \Psi) \\ Z(unit, \Psi) &= 1 \\ Z(empty, \Psi) &= 0 \end{aligned}$$

By caching these computations for each subexpression of D, these equations give a way for computing $Z(D, \Psi)$ in time proportional to $|D|$. These equations are analogous to the inside algorithm used in statistical parsing.

The Viterbi Algorithm We define $\Psi^*(D, \Psi)$ as follows:

$$\Psi^*(D, \Psi) = \min_{\rho \in F(D)} \Psi(\rho)$$

We can compute Ψ^* using the following equations:

$$\begin{aligned} \Psi^*(case(z, D_1, D_2), \Psi) &= \min(\Psi(z) + \Psi^*(D_1, \Psi), \Psi(D_2, \Psi)) \\ \Psi^*(factor(D_1, D_2), \Psi) &= \Psi^*(D_1, \Psi) + \Psi^*(D_2, \Psi) \\ \Psi^*(unit, \Psi) &= 0 \\ \Psi^*(empty, \Psi) &= +\infty \end{aligned}$$

Note that these computations work on log-probabilities. And the induction is proved on the size of $|D|$.

The Inside-Outside Algorithm The value $Z(D, \Psi)$ is the "inside" value associated with D. Intuitively, the outside value of a node in a CFD is the total weight of the "context" in which that node appears. Although the definition of context is subtle (McAllester et al., 2004), the equation for computing outside values are rather straightforward.

A node is *open* if it does contain variables and *closed* otherwise. Outside values are only defined for open nodes. We now use $O(D', D, \Psi)$ to denote the outside value of D' for $D' \preceq D$ and D' is open.

When $D' = D$, $O(D', D, \Psi) = 1$; otherwise:

$$
\begin{aligned}
O(D', D, \Psi) &= \sum_{case(z, D', D") \preceq D} O(case(z, D', D"), D, \Psi) e^{-\Psi(z)} \\
&+ \sum_{case(z, D", D') \preceq D} O(case(z, D", D'), D, \Psi) \\
&+ \sum_{factor(D', D") \preceq D} O(factor(D', D"), D, \Psi) Z(D", \Psi) \\
&+ \sum_{factor(D", D') \preceq D} O(factor(D", D'), D, \Psi) Z(D', \Psi)
\end{aligned}
$$

Observe the right-hand-side of the equation, for the last two items, intuitively, if we take $factor(D', D")$ as the "parent" node of D' and $D"$, it is fairly straightforward to understand. If taking variable z as the "parent" node of D' and $D"$, then the first item corresponds to the case of z and D' b being in the same sub-branch and its "sibling" set being empty; the second item corresponds to the case D' in the subbranch with no z, so those variables that "dominate" it only come from the "upper-part" of z.

From the above analysis, we see the main inference algorithms on CFD still follow the same induction form as the classic algorithms. Once the inside value of every node has been computed, these equations allow the outside values of open nodes to be computed from the top down. This top-down calculation can be done in time proportional to the number of nodes. Finally, we can compute $P(z = 1|D, \Psi)$ as follows:

Theorem

$$
\begin{aligned}
P(z = 1|D, \Psi) &= \frac{Z(D, \Psi, \emptyset[z := 1]}{Z(D, \Psi)} \\
Z(D, \Psi, \emptyset[z := 1] &= \\
\sum\nolimits_{case(z, D', D") \preceq D} &(O(case(z, D', D"), D, \Psi) e^{-\Psi(z)} Z(D', \Psi))
\end{aligned}
$$

10.3.6 CFD Models and Dependency Graphs in Semiring Parsing

From discussion above, there are still some similarities between CFD modeling and the dependency graph structure in the semiring parsing system:

First, dependency graph is essentially a directed AND-OR graph: the set of nodes consists of all derivable items and the directed edges denote the dependency

(or derivable) relations between nodes (or items). This structure follows the traditional AND-OR graph structure and we can calculate the inside/outside values for each node by applying the classic Inside-Outside algorithms to the structure. More general, some generic algorithms can be applied to this structure in the semiring framework, which we are interested in.

Compared with the dependency graph structure, CFD model gives a more general formalism for Markov Random Fields. It represent the structure by defining the "case expressions" and "factor expressions", which seems a more concise formalism. And as an example, PCFG can be nicely encoded by this formalisms and keep the same complexity for the general PCFG inference algorithms. However, in CFD for PCFG, we still have to list all the variables (i.e., phrase variable, branch variables and terminal variables), which corresponds to the derivable items and rules in dependency graph representation, and based on this to set up a correspondence between variables and Boolean variables. So the relations ("case" or "factor") between variables revealed by the expressions have to be set up in advance, which is essentially the same as that in dependency graph.

10.4 Summary of this Section

In this section, we presented more structured probabilistic formalisms beyond dependency graph for PCFG. Feature forests can incorporate a richer set of features without any independence assumptions. MK packed representations and CFD are using a set of boolean vectors to take over the calculations. All these formalisms are basically associated with some graph structure and based on the graph structure doing some dynamic programming. Taking parsing as the one of main tasks, the calculations of Viterbi value and inside/outside values are almost identical for these formalisms. All these provide a possibility for generalizing parsing and parameter estimations into the semiring parsing system, and give us a wider perspective beyond PCFGs, dependency graph of producing the parses.

11 Summary and Future Work

In this report, we first recalled some basics of formal language theory, in particular, regular languages, context free languages and CKY/Earley's/GHR parsing algorithms. To uncover the algebraic foundation of semiring parsing, we investigated in detail the concepts of formal power series and the semirings, and the closure properties of rational power series and algebraic power series in particu-

lar, which correspond to the regular languages and context-free languages respectively. The probe of algebraic properties helps us view the statistical parsing in a more general and unified perspective. In the framework of power series, parsing can be taken to be the computation of the coefficients for the elements of a free monoid. Semiring parsing is based on the general view and by assigning different semantics to the operations in parsing to achieve the unified representation of parsing across diverse tasks.

In practice, we have to consider how to efficiently organize the exponentially many parses and how to effectively introduce good search strategies in deduction without loss of system generality. Shared forests structure has been proposed and taken as a standard way of organizing all the possible parses for a particular input sentence. And some possible search strategies have been explored to speed up the statistical parsing process. More general formalisms beyond PCFGs have been presented, which either incorporate a much richer set of features into the packed representations or take over the calculations to a set of Boolean vectors. All these still happen on graph structures with partial order posed on the nodes and are still using the classic parsing and learning algorithms.

By investigating this area in depth, in the future work, we are consider the following aspects:

- Incorporate some good search strategies into the general semiring parsing system and customize the search strategy for some particular semiring. We will endeavor to find some "good" parsing strategies that fit into the general framework with maximum flexibility.

- Apply a more general formalism to the semiring parsing, such as feature forest structure. One thing is to see if there exists an item-based-description-like representation to produce the feature forest; furthermore try to assign the semantics to the parameter estimation process using semirings (Eisner, 2001).

- Item-based descriptions provide a way to unify a variety of parsing algorithms on various formalisms. We may try a wider possible applications of item-based descriptions, such as bilingual text alignments in machine translation and other NLP tasks involving mapping and transductions.

- Parsing as a powerful vehicle for syntactic structure analysis can be applied in many different tasks and areas, such as the sequence analysis of nucleic acids in bioinformatics. We will do more experiments on semiring parsing

system, one hand it is for testing its applicability; on the other hand, for motivating more improvements and more applications by more experiments.

- The closure properties of rational power series enable much of the power and flexibility of the FSM toolkit. We will try to explore the properties of power series for other language formalisms, such as algebraic power series for context-free languages, to see if there exist elegant algebraic properties that can lead to a powerful toolkit that can manipulate CFGs.

References

Bar-Hillel, Y., M. Perles, and E. Shamir. 1964. On formal properties of simple phrase structure grammars. In Y. Bar-Hillel, editor, *Language and Information:Selected Essays on Their Theory and Applications*. Addison-Wesley, chapter 9, pages 116–150.

Barthelemy, Francois, Pierre Boullier, Philippe Deschamp, and Eric de la Clergerie. 2001a. Guided parsing of range concatenation languages. In *Proceedings of ACL 2001*.

Barthelemy, Francois, Pierre Boullier, Philippe Deschamp, and Eric Villemonte de la Clergerie. 2001b. Guided parsing of range concatenation languages. In *Meeting of the Association for Computational Linguistics*, pages 42–49.

Berger, Adam L., Vincent J. Della Pietra, and Stephen A. Della Pietra. 1996. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71.

Berstel, Jean. 1979. *Transductions and Context-Free Languages*. Teubner Studienbucher.

Billot, S. and B. Lang. 1989. The structure of shared forests in ambiguous parsing. In *Proceedings of ACL 1989*.

Bouchard, Guillaumi. 2003. The trade-off between generative and discriminative classifiers.

Boullier, Pierre. 2000. Range concatenation grammars. In *Proceedings of the Sixth International Workshop on Parsing Technologies(IWPT 2000)*, pages 53–64.

Boullier, Pierre. 2003a. Guided Earley parsing. In *IWPT 2003*.

Boullier, Pierre. 2003b. Supertagging: A non-statistical parsing-based approach. In *IWPT 2003*.

Bouloutas, A., G. W. Hart, and M. Schwartz. 1991. Two extensions of the Viterbi algorithm. *IEEE transactions on information theory*, 37(2):430–436.

Caraballo, Sharon A. and Eugene Charniak. 1998. New figures of merit for best-first probabilistic chart parsing. In *Proceedings of ACL 1998*.

Carnie, Andrew. 2002. *Syntax: A Generative Introduction*. Blackwell Publishing.

Charniak, E., S. Goldwater, and M. Johnson. 1998. Edge-based best-first chart parsing. In *Proceedings of the 6th Workshop on Very Large Corpora*.

Chiang, David. 2004. Mildly context sensitive grammars for estimating maximum entropy parsing models. In *Proceedings of FGVienna: The 8th Conference on Formal Grammar*.

Chomsky, N. and M.P. Schutzenberger. 1963. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*. North-Holland, Amsterdam, pages 118–161.

Church, K. and R. Patil. 1982. Coping with syntactic ambiguity or how to put the block in the box on the table. *America Journal of Computational Linguistics*, 8(3-4).

Collins, Michael. 1997. Three generative, lexicalised models for statistical parsing. In *Proceedings of ACL 1997*.

Collins, Michael. 2001. Parameter estimation for statistical parsing models: Theory and practice of distribution-free methods. In *IWPT 2001*.

Collins, Michael. 2002. Discriminative training methods for hidden markov models: theory and experiments with the perceptron algorithm. In *EMNLP 2002*.

Collins, Michael. 2004. Discriminative reranking for natural language parsing(submission version). *Computational Linguistics*.

Earley, Jay. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Eisner, Jason. 2001. Expectation semirings: Flexible EM for finite-state transducers. In Gertjan van Noord, editor, *Proceedings of the ESSLLI Workshop on Finite-State Methods in Natural Language Processing*.

Freund, Yoav and Robert E. Schapire. 1999. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771–780.

G. Rozenberg, A. Salomaa. 1997. *Handbook of formal languages*. Berlin ; New York : Springer.

Geman, Stuart and Mark Johnson. 2002. Dynamic programming for parsing and estimation of stochastic unification-based grammars. In *Proceedings of ACL 2002*.

Goodman, Joshua. 1998. *Parsing Inside-Out*. Ph.D. thesis, Harvard University.

Goodman, Joshua. 1999. Semiring parsing. *Computational Linguistics*, 25(4):573–605.

Griffiths, T. V. and S. R. Petrick. 1987. On the relative efficiencies of context-free grammar recognizer. *Computational Linguistics*, 8(5):289–300.

Guingne, Franck and Florent Nicart. 2003. Finite state lazy operation in NLP. In *Implementation and Application of Automata, 7th International Conference, CIAA 2002*, volume 2608 of *Lecture Notes in Computer Science*. Springer.

Jebara, Tony. 2004. *Maching Learning: Discriminative and Generative*. Kluwer Academic Publishers.

Jelinek, Frederick and John D. Lafferty. 1991. Computation of the probability of initial substring generation by stochastic context-free grammars. *Computational Linguistics*, 17(3):315–323.

John E. Hopcroft, Jeffrey D. Ullman. 1979. *Introduction to automata theory, languages, and computation*. Addison-Wesley Springer-Verlag.

Johnson, Mark. 2001. Joint and conditional estimation of tagging and parsing models. In *Proceedings of ACL 2001*.

Johnson, Mark. 2004. On discriminative approaches to statistical parsing and on statistical parsing and speech recognition.

Kanal, Laveen and Vipin Kumar. 1988. *Search in Artificial Intelligence*. Springer-Verlag.

Klein, Dan and Christopher D.Manning. 2001. Parsing and hypergraphs. In *WPT-2001*.

Klein, Dan and Christopher D. Manning. 2001. An O(n^3) agenda-based chart parser for arbitrary probabilistic context-free grammars. Technical Report dbpubs/2001-16, Standford University.

Klein, Dan and Christopher D. Manning. 2003. A* parsing: Fast exact Viterbi parse selection. In *Proceedings of the HLT-NAACL 2003*.

Knuth, D. E. 1965. On the translation of language from left to right. *Information and Control*, 8:607–639.

Knuth, D. E. 1977. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5.

Kuich, Werner and Arto Salomaa. 1985. *Semirings, Automata, Languages*. Springer-Verlag.

Lang, Bernard. 1974. Deterministic techniques for efficient non-deterministic parsers. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*.

Lang, Bernard. 1991. Towards a uniform formal framework for parsing. In M. Tomita, editor, *Current Issues in Parsing Technology*. Kluwer Academic Publishers, pages 153–171.

Lang, Bernard. 1994. Recognition can be harder than parsing. *Computational Intelligence*, 10:486–494.

Lari, K. and S. J. Young. 1990. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56.

L.Graham, Susan, Michael A. Harrison, and Walter L.Ruzzo. 1980. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462.

M., Mohri, Pereira F. C. N., and Riley M. 1998. A rational design for a weighted finite-state transducer. *Lecture Notes in Computer Science*, 1436.

M.Aji, Srinivas and Robert J.McEliece. 2000. The generalized distributive law. *Transactions on Information Theory*, 46(2).

Malouf, Robert. 2002. A comparison of algorithm for maximum entropy parameter estimation. In *Proceedings of CoNLL-2002*.

Manning, Christopher D. and Hinrich Schutze. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press.

Maxwell III, John T. and Ronald M. Kaplan. 1981. A method for disjunctive constraint satisfaction. In Masaru Tomita, editor, *Current Issues in Parsing Technology*. Kluwer Academic Publishers, Dordrecht, pages 173–190.

McAllester, David, Michael Collins, and Fernando Pereira. 2004. Case-factor diagrams for structured probabilistic modeling. In *UAI 2004*.

Miyao, Yusuke. 2003. Feature forest models for syntactic parsing. In *Japanese-German Workshop on Natural Language Processing: NLP for Information Management and Semantic Web*.

Miyao, Yusuke and Jun'ichi Tsujii. 2002. Maximum entropy estimation for feature forests. In *Proceedings of HLT 2002*.

Mohri, Mehryar. 1996. On some applications of finite-state automata theory to natural language processing. *Journal of Natural Language Engineering*, 1(1):1–20.

Mohri, Mehryar. 1997. Finite-state transducers in language and speech processing. In *ACL97*.

Mohri, Mehryar. 2002. Semiring framework and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350.

Mohri, Mehryar, Fernando C. N. Pereira, and Michael Riley. 2000. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231:17–32.

Mohri, Mehryar and Michael Riley. 2002. Weighted finite-state transducers in speech recognition (tutorial). In *In Proceedings of the International Conference on Spoken Language Processing 2002 (ICSLP '02)*.

Moore, Robert. C. 2000. Improved left-corner chart parsing for large context-free grammars. In *IWPT 2000*.

Nederhof, M. J. 1993. Generalized left-corner parsing. In *Proceedings of the Sixth Conference of the European Chapter of ACL*.

Nederhof, M. J. 1994. An optimal tabular parsing algorithm. In *Proceedings of ACL 1994*.

Nederhof, M.-J. 1996. Introduction to finite-state techniques.

Nederhof, M. J. 1998. Context-free parsing through regular approximation. In *Proceedings of the International Workshop on Finite Methods in NLP*.

Nederhof, M. J. 2000. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1):17–44.

Nederhof, M. J. 2003. Weighted deductive parsing and Knuth's algorithm. *Computational Linguistics*, 29(1):135–143.

Nederhof, M. J., Anoop Sarkar, and Giorgio Satta. 2001. Prefix probabilities from stochastic tree adjoining grammar. In *Proceedings of ACL 2001*.

Nederhof, Mark-Jan and Giorgio Satta. 2003. Probabilistic parsing as intersection. In *Proceedings of the 8th International Workshop on Parsing Technologies(IWPT 2003)*.

Ng, Andrew Y. and Michael Jordan. 2002. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Neural Information Processing Systems 14*.

Pereira, Fernando C. N. and Michael D. Reiley. 1997. Speech recognition by composition of weighted finite automata. In *Finite State Devices in Natural Language Processing*. MIT Press.

Ratnaparkhi, Adwait. 1997. A simple introduction to maximum entropy models for natural language precessing. Technical Report IRCS-97-08, Institute for Research in Cognitive Science.

Ratnaparkhi, Adwait. 1999. Learning to parse natural language with maximum entropy models. *Machine Learning*, 34:151–178.

Rubinstein, Y. D. and T. Hastie. 1997. Discriminative vs. informative learning. In *KDD 1997*.

S. M. Shieber, Y. Schabes, F. C. N. Pereira. 1995. Principles and implementations of deductive parsing. *Journal of Logic Programming*, 24(1-2):3–36.

Salomma, Arto and Matti Soittola. 1978. *Automata-Theoretic Aspects of Formal Power Series*. Springer-Verlag Berlin Heidelberg New York.

Schabes, Yves. 1991. Polynomial time and space shift-reduce parsing of arbitrary context-free grammars. In *Meeting of the Association for Computational Linguistics*, pages 106–113.

Schutzenberger, M. P. 1961. On the definition of a family of automata. *Information Control*, 4:245–270.

Sippu, Seppo and Eljas Soisalon-Soininen. 1988. *Parsing Theory*, volume 1: Languages and Parsing. Springer-Verlag Berlin Heidelberg New York.

Stolcke, Andreas. 1993. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. Technical Report TR-93-065, International Computer Science Institute.

Tendeau, F. 1998. Computing abstract decoration of parse forests using dynamic programming and algebraic power series. *Theoretical Computing Science*, 199:145–166.

Tomita, Masaru. 1987. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1-2):31–46.

Tsuruoka, Yoshimasa and Junichi Tsujii. 2004. Iterative CKY parsing for probabilistic context-free grammar. In *IJCNLP 2004*.

Appendix A Algorithms in PCFGs

Note: Here we only consider the case of CNF grammars, which only have unary and binary rules as defined above.

The Probability of a String

- Using *Inside Probabilities:*
  The inside probability $\beta_j(p,q)$, $\beta_j(p,q) = P(w_{pq} \mid N^j_{pq}, G)$, is the total probability of generating words $w_p \cdots w_q$ given that one is starting off with the nonterminal N^j.

  An efficient way to calculate the total probability of a string is by the *inside algorithm*, a dynamic programming algorithm based on the inside probabilities:

  $$(1) \qquad P(w_{1m} \mid G) = P(w_{1m} \mid N^1_{1m}, G) = \beta_1(1,m)$$

  Using the following recurrence relation, inside probabilities can be efficiently calculated bottom up:

  Base case: $\beta_j(k,k) = P(N^j \rightarrow w_k \mid G)$ (Probability of rule $N^j \rightarrow w_k$)
  Induction:

  $$\beta_j(p,q) = P(w_{pq} \mid N^j_{pq}, G) = \sum_{r,s} \sum_{d=p}^{q-1} P(N^j \rightarrow N^r N^s)\beta_r(p,d)\beta_s(d+1,q)$$

  Above, things are firstly divided up using the chain rule, then based on the context-free assumptions of PCFG, we rewrote the result using the definition of the inside probability.

- Using *Outside Probabilities:*
  We can also calculate the probability of a string via the use of the outside probabilities. For any $k, 1 \leq k \leq m$,

  $$(2) \qquad P(w_{1m} \mid G) = \sum_j \alpha_j(k,k) P(N^j \rightarrow w_k)$$

  Outside probability $\alpha_j(p,q)$, $\alpha_j(p,q) = P(w_{1(p-1)}, N^j_{pq}.w_{(q+1)m} \mid G)$, is the total probability of beginning with the start symbol N^1 and generating the nonterminal N^j_{pq} and all the words outside $w_p \cdots w_q$.

The outside probabilities are calculated top down. And the inductive calculation of outside probabilities requires reference to inside probabilities.

Base case: The base case is the probability of the root of the tree being nonterminal N^i with nothing outside it.

$$\alpha_1(1,m) = 1 \quad \alpha_j(1,m) = 0 \text{ for j } \neq 1$$

Inductive case:

$$\alpha_j(p,q) = \sum_{f,g \neq j} \sum_{e=q+1}^{m} \alpha_f(p,e) P(N^f \rightarrow N^j N^g) \beta_g(q+1.e)$$

$$+ \sum_{f,g} \sum_{e=1}^{p-1} \alpha_f(e,q) P(N^f \rightarrow N^g N^j) \beta_g(e,p-1)$$

- *Combining Inside and Outside Probabilities:*
  As with an HMM, we can form a product of the inside and the outside probabilities:
  $$\alpha_j(p,q)\beta_j(p,q) = P(w_{1m}, N^j_{pq} \mid G)$$
  And the probability of the sentence and that there is some constituent spanning from word p to q is given by:
  $$P(w_{1m}, N_{pq} \mid G) = \sum_j \alpha_j(p,q)\beta_j(p,q)$$
  So equation(1) is equivalent to $\alpha_1(1,m)\beta_1(1,m)$ and makes use of the root node, while equation(2) is equivalent to $\sum_j \alpha_j(k,k)\beta_j(k,k)$ and makes use of the fact that there must be some preterminal N^j above each word w_k.

Finding the most likely parse for a sentence

Similar to the Viterbi algorithm to HMMs, here we wish to find the most probable parse tree for the given string in PCFG. By the notation: $\delta_i(p,q) =$ the highest inside probability parse of a subtree N^i_{pq}, we use dynamic programming to calculate the most likely parse as follows:

1. Initialization
   $$\delta_i(p,p) = P(N^i \rightarrow w_p)$$

2. Induction

$$\delta_i(p,q) = \max_{1\leq j,k\leq n, p\leq r<q} P(N^i \rightarrow N^j N^k)\delta_j(p,r)\delta_k(r+1,q)$$

Store backtrack

$$\psi_i(p,q) = \text{argmax}_{(j,k,r)} P(N^i \rightarrow N^j N^k)\delta_j(p,r)\delta_k(r+1,q)$$

3. Termination and path readout (by backtracking). The probability of the most likely parse rooted in the start symbol is:

$$P(\hat{t}) = \delta_1(1,m)$$

By observation, different from the inside algorithm, this time we find the most probable one by taking the maximum operation instead of summing over all such rules.

This Viterbi-style algorithm results in an $O(m^3 n^3)$ PCFG parsing algorithm, where m is the length of the sentence, and n is the number of nonterminals in the grammar. And it is possible for there not be a unique maximum. We assume in such cases the parser just choose one maximal parse at random.

Another question of interest is **"How can we choose rule probabilities for the grammar G that maximize the probability of a sentence:"**
$\text{argmax}_G P(w_{1m} \mid G)$?. It essentially involves the problem of how to train a PCFG.

Training a PCFG The idea of training a PCFG is grammar learning or grammar induction, but only in a certain limited sense. Here we assume the structure of the grammar is given in advance. Training the grammar comprises simply a process that tries to find the optimal probabilities to assign to different grammar rules within this architecture.

As in the case of HMMs, we construct an EM training algorithm, the *Inside-Outside algorithm*, which allows us to train the parameters of a PCFG on unannotated sentences of the language. The basic assumption is that a good grammar is one that makes the sentence in the training corpus likely to occur, and hence we seek the grammar that maximize the likelihood of the training data. The training can be extended from a single sentence to the more realistic situation of a large training corpus of many sentences, by assuming independence between sentences.

To determine the probability of rules, what we would like to calculate is :

$$(3) \quad \hat{p}(N^j \rightarrow \zeta) = \frac{C(N^j \rightarrow \zeta)}{\sum_\gamma C(N^j \rightarrow \gamma)}$$

where C(.) is the count of the number of times that a particular rule is used. If parsed corpora are available, we can calculate these probabilities directly(Empirical Relative Frequency). If, as is more common, a parsed training corpus is not available, then we cannot compute the relative frequencies, so we instead use an iterative algorithm to determine improving estimates.

Based on the Maximum Likelihood assumption and equation (3), we want:

$$P(N^j \rightarrow N^r N^s) = \frac{E(N^j \rightarrow N^r N^s, N^j \text{ used})}{E(N^j \text{ used})}$$

We will use it as our re-estimation formula. We have already solved how to calculate $P(N^1 \Rightarrow^* w_{1m}$; let us call this probability π. Then the estimation for how many times the nonterminal N^j is used in the derivation is:

$$E(N^j \text{ is used in the derivation } = \sum_{p=1}^{m} \sum_{q=p}^{m} \frac{\alpha_j(p,q)\beta_j(p,q)}{\pi}$$

And the estimate for how many times a particular rule is used in the derivation can be found by summing over all ranges of words that the node could dominate:
$E(N^j \rightarrow N^r N^s, N^j \text{ used})$

$$= \frac{\sum_{p=1}^{m-1} \sum_{q=p+1}^{m} \sum_{d=p}^{q-1} \alpha_j(p,q) P(N^j \rightarrow N^r N^s) \beta_r(p,d) \beta_s(d+1,q)}{\pi}$$

Similarly, for the pre-terminals:

$$\hat{p}(N^j \rightarrow w^k) = \frac{\sum_{h=1}^{m} \alpha_j(h,h) P(w_h = w^k) \beta_j(h,h)}{\sum_{p=1}^{m} \sum_{q=p}^{m} \alpha_j(p,q) \beta_j(p,q)}$$

Given some grammar topology and some initial probability estimates for rules, we compute the expectation of how often each rule was used(this computation is mainly based on the inside probabilities). These expectations are then used to refine our probability estimates on rules, so that the likelihood of the training corpus given the grammar is increased.

Now let us assume that we have a set of training sentences $W = (W_1, \cdots, W_\omega$. If we assume that the sentences in the training corpus are independent, then the likelihood of the training corpus is just the product of the probabilities of the sentences in it according to the grammar. Therefore, in the reestimation process, we can sum the contributions from multiple sentences.

This estimation process is called *Inside-Outside algorithm*. It is to repeat this process of parameter re-estimation until the change in the estimated probability of the training corpus is small.

If G_i is the grammar(including rule probabilities) in the i^{th} iteration of training, then we are guaranteed that the probability of the corpus according to the model will improve or at least get no worse:

$$P(W \mid G_{i+1}) \geq P(W \mid G_i)$$

This algorithm can give a solution of local maxima.