The Earley Parsing Cookbook

Jeremy Dohmann

March 26, 2019

Introduction

Abstract

CONTENTS

Abstract 1

Chapter 1: Greedily finding best parses for context-free grammars as a problem in logical deduction

Abstract

Parsing is one of the most fundamental tasks in formal language theory. The first algorithm for parsing general context free languages was developed by Jay Earley in 1968 [1]. Though much of the field has been concerned with statistical methods in parsing, there is still a case to be made that developments in strictly symbolic parsing are vital to computational linguistics and other related tasks in applied computation. Many data processing tasks have sufficiently sophisticated structure to warrant CFG parsing, but they often lack the data necessary to rely solely on statistical parsing methods. Additionally, applied computation tasks almost always involve imputing structure upon highly noisy data.

This paper focuses on greedily obtaining best parses for noise-skipping Earley parsers. This task is vital in circumstances in which the input data stream is noisy and corpora are not available to guide statistical parsing algorithms, but the user still has a sense of what constitutes a "good parse". We will explore what class of utility functions can be validly used to greedily compute best parses. We will do so by unifying this approach with semiring parsing and parsing as deductive logic. Two important approaches pioneered by [2] and [3].

Put formally in terms of deductive logic, I will address the following technical problem: given a consequent B, and an inference rule showing its truth:

$$\text{RULE R}\ \frac{A_1 \quad ... \quad A_k}{B}\ \langle side\ conditions \rangle$$

If we have a utility function $f() : \mathbb{R}^N \rightarrow \mathbb{R}$, a way of describing proofs as a vector of attributes $X(B) \in \mathbb{R}^N$ where each entry of $X(B)$ is in $\mathbb{R}$ (e.g. the log-probability of the proof based on the probabilities of the inference rules in the logic system, or e.g. number of axioms used in the proof, or the number of inference rules used in total), and a function $g() : \mathbb{R}^{N \times k+1} \rightarrow \mathbb{R}^N$ defining the relationship between the attributes of $X(B)$, the attributes of the rule used to derive it $X(RuleR)$, and the attributes of its antecedents $X(A_k)$, under what circumstances can we greedily compute the utility of a proof for a given span of tokens provided only the attribute vectors for the *best* proofs of its antecedents? Put formally: If

(1) any given proof of B has attributes $X(B)$ determined by the attributes of the proofs of its antecedents $X(A_i)$ and inherent attributes of the inference rule X_{RuleR}, determined according to a combination function
$X(B) = g(X(A_1), ..., X(A_k), X_{RuleR})$ where $g : \mathbb{R}^{N \times k+1} \rightarrow \mathbb{R}^N$

(2) we can rank proofs by their attributes using a utility function $f(X) : \mathbb{R}^N \rightarrow \mathbb{R}$,

For which choices of combination function $g(\cdot)$ and utility function $f(\cdot)$ can we greedily determine the utility and attributes of the best proof of B given just the attributes $(X(A_i))$ of the best proofs of its antecedents and the inherent attributes $(X(RuleR))$ of the inference rule used in its derivation?

As mentioned, this approach is aimed at developing criteria for which utility functions can be used in noise-skipping Earley parsing, but my conception of the problem will be a deductive-logic based approached introduced by [3].

Viewed in the Earley context, the question becomes, when can we greedily determine the best tree associated with parsing a span i, j as constituent type B given the best trees associated with its parsed subspans?

In addition to exploring the conditions controlling the properties of valid $f(\cdot)$ and $g(\dot{})$ I will also address the algorithmic requirements of this model - in particular I will show the pitfalls that arise from poor choice of underlying data structure in your Earley chart and show that priority queues ranked in decreasing order of start index for a state can ensure correct greedy calculation of optimal derivations as long as the grammar contains no cycles of unary productions (e.g. $A \rightarrow B$ and $B \rightarrow C$).

CONTENTS

Abstract		1
Contents		1
1	Introduction	1
2	Optimal solutions	2
3	An example	3
3.1	Breaking optimality	3
4	Optimality of linear utility functions	3
5	Optimality as a semiring	4
5.1	Distributivity of $\otimes$	4
6	Preventing out-of-order processing	5
6.1	How choice of data structure impacts out-of-order processing	6
6.1.1	Stack ADTs	6
6.1.2	Priority queue ADTs	7
6.1.3	Topological sorting	8
6.1.4	Queue ADTs	9
References		9

1 Introduction

The Earley parser is a parsing technique for both recognizing string membership in a (potentially ambiguous)

context-free language (CFL) and for describing the possible parses (derivations) of that string under the rules of the rewrite system (RWS) description of that CFL.

[3] provides a unified framework to view the Earley parser, and general parsing algorithms as deductive logic proofs.

The general form of an inference rule in this framework is

$$\frac{A_1 \quad ... \quad A_k}{B} \ \langle side\ conditions \rangle$$

Where A_i are antecedents and B is the consequent, which can be *deduced* given that its antecedents are true and the side conditions for the inference rule are met.

To quote [3]:

> Given a grammatical deduction system, a *derivation* of a formula B from assumptions $A_1, ..., A_m$ is, as usual, a sequence of formulas $S_1, ..., S_n$ such that $B = S_n$ and for each S_i, either s_i is one of the A_j, or S_i is an instance of an axiom, or there is a rule of inference R and formulas $S_{i_1} ... S_{i_k}$ with $i_1, ... i_k < i$ such that for appropriate substitutions of terms for the metavariables in R, $S_{i_1}, ..., S_{i_k}$ match the antecedents of the rule, S_i matches the consequent, and the rule's side conditions are satisfied.

This definition corresponds to the usual definition of a *derivation* in deductive logic, and corresponds to the intuitive notion of a parse tree in syntax. Thus in Shieber's methodology, theorems correspond to statements "span i,j can be parsed to a nonterminal A" and proofs correspond to parse trees.

An important point to note is that a derivation of B may be ambiguous for context-free grammars. Ambiguity may either arise because multiple inference rules can be used to derive B:

$$\frac{A_1 \quad ... \quad A_k}{B} \ \langle side\ conditions \rangle$$

$$\frac{C_1 \quad ... \quad C_m}{B} \ \langle side\ conditions \rangle$$

or for a derivation of B one of its antecedents can be derived via multiple inference rules (or a combination thereof).

In linguistic terms, a constituent can be described by an item $[A, i, k]$ which says the elements in span i, k can be categorized as a constituent of type A and the parse trees anchored at A and explaining span i, k correspond to the, perhaps multiple, derivations of item $[A, i, k]$ via the inference rules of the deductive system/via the RWS rules of the grammar.

Thus, it is important to note that $[A, i, k]$ can be considered to be a **theorem** (i.e. span i, k can be derived from nonterminal A) to which there may be many **proofs** (i.e. the corresponding deductive derivations/chain of RWS rules applied to generate the span from A).

In general, inference rules will take the form

$$\frac{[B, i, j] \quad [C, j, k]}{[A, i, k]} \ \mathrm{A} \rightarrow BC \tag{1}$$

Where the items of the deductive system are problem statements (e.g. derive i, j from B) and the side conditions are the existence of rules in the grammar permitting such a derivation.

Clearly if there are r possible derivations of $[B, i, j]$ and s possible derivations of $[C, j, k]$ then there are $r \cdot s$ possible derivations of $[A, i, k]$.

I am particularly interested applications to noisy parsing where spans may be skipped in the derivation. For example this may permit derivations of the form:

$$\frac{[B, i, j'] \quad [C, j, k]}{[A, i, k]} \ \mathrm{A} \rightarrow BC \wedge j' \leq j \tag{2}$$

where $[B, i, j - 3]$ and $[C, j, k]$ can be merged, skipping 3 tokens in the process.

2 Optimal solutions

As stated previously, the theorem $[A, i, k]$ corresponds to stating there exists at least one parse tree of root type A which generates span i, k. Since there may be many such trees we will index them as $T^j(A, i, k)$ where $T^j(A, i, k)$ is the j^{th} way to parse i, k as type A. In deductive logic terms, once more, each tree corresponds to a unique proof of the theorem $[A, i, k]$. Let's say that any given tree (alternatively, proof) has some vector X_T of attributes associated with it based on its structure, e.g. number of constituents contained in it, number of tokens explained, number of tokens skipped, etc.

Given a derivation

$$\textsc{Rule R}\ \frac{[B, i, j] \quad [C, j, k]}{[A, i, k]} \ \mathrm{A} \rightarrow BC \tag{3}$$

a parse tree $T(A, i, k)$ composed of subtrees $T(C, i, j)$ and $T(B, j, k)$ has attributes $X(A)$ which are a function of the attributes of subtrees $T(C, i, j)$ and $T(B, j, k)$, plus some additional attributes, $X(Rule\ R)$, which $[A, i, k]$ contributes irrespective of which derivations of B and C we choose.

Where $X(Rule\ R)$ denotes inherent attributes of this derivation.

Therefore $X(A)$ is derviable from $X(B)$, $X(C)$, and $X(Rule\ R)$ via some function $g(X(B), X(C), X(Rule\ R)$ which I will refer to as the "combination" function.

That is, the attributes under study are natural insofar as they are compositional in the subtrees of a given tree. Put into logical terms, the attributes of a proof are a function of the proofs of the antecedents plus some inherent properties of the inference rule being applied to achieve the consequent.

I will study utility functions $f(X_T)$ that perform rankings over all $T^j(A, i, k)$ for a given $[A, i, k]$. I will call $T(A, i, k)$ the best tree associated with $[A, i, k]$ for a given $f(\cdot)$.

We would like to determine which choices of $g(\cdot)$ (combination functions) and $f(\cdot)$ (utility functions) satisfy an

important condition I dub the "optimality" condition which relates whether a system of functions $g(\cdot)$ and $f(\cdot)$ can be used to greedily construct the best parses at each step of the derivation process.

Definition 1. Optimality condition $f(\cdot)$ and $g(\cdot)$ satisfy the optimality condition if and only if for all derivations of the form (1) we can calculate the optimal $T(A, i, k)$ by choosing any one of the optimal $T(B, i, j)$ and any one of the optimal $T(C, j, k)$ and merging their attributes via $X(A) = g(X(B), X(C), X(Rule\ R))$, where $X(Rule\ R)$ is the inherent attributes of deriving $[A, i, k]$ by the inference rule (1). In the case of non unique optima for the subproblems we should be able to break ties arbitrarily

This question is an important one in Earley parsing. If there is an application in which we can return many possible completed sentential forms $[S, i, j] \in \mathbb{S}$ found throughout a text and are interested only in the best parse tree amongst those possible forms. It would be ideal to extract the best possible parse of span i, j without explicitly extracting the whole forest from each of the states in $[S, i, j] \in \mathbb{S},$. This is especially important in noise-skipping parsing, because the potential for many partial solutions to proliferate can inundate the system with parses of radically different utility values.

To do so would require that we maintain, for every single completed state $[A, i, k]$, a reliable calculation of the *best* tree deriving that completed state. That is, we would need to be able to easily calculate $T(A, i, k)$ at the time that $[A, i, k]$ is completed. Our suggestion here is the best way to do so is to choose $f(\cdot)$ and $g(\cdot)$ such that they satisfy the optimality condition and therefore. Thus, when completing $[A, i, k]$ the utility of its best derivation is equal to $f(X(A))$ where $X(A) = g(X(B), X(C), X(Rule\ R)$.

3 An example

For any $g(\cdot)$ which is nonincreasing at any point in its domain you can find a reasonable $f(\cdot)$ for which $g(\cdot)$ fails both strong and weak optimality.

3.1 Breaking optimality

A simple example is illustrated below:

Imagine $[B, i, j]$ and $[C, j + 1, k]$ are unified via the inference rule

$$\frac{[B, i, j'] \quad [C, j, k]}{[A, i, k]}\ \mathrm{A} \rightarrow BC \wedge j' \leq j \tag{4}$$

Furthermore, imagine that the attributes vector contains the following values: number of tokens explained, number of tokens skipped, number of subsequences skipped, and number of constituents. Clearly $X(Rule\ R)$, which is invariant under the choice of derivation for $[B, i, j]$ and $[C, j+1, k]$, is $[0, 1, 1, 1]$ because it skipped a single token in the merge, and therefore also a single subsequence, and contributes a single new constituent of its own (namely, itself).

If there are two solutions to $[B, i, j]$

(1) $T^1(B, i, j)$ with attributes $[5, 3, 1, 2]$ with all the skipped tokens forming a contiguous block on the right-boundary of $[i, j]$
(2) $T^2(B, i, j)$ with attributes $[5, 3, 1, 2]$ with the skipped tokens somewhere in the interior of span $[i, j]$

and $[C, j + 1, k]$ has similar solutions:

(1) $T^1(C, j + 1, k)$ with attributes $[2, 2, 1, 1]$ with all the skipped tokens forming a contiguous block on the left-boundary of $[i, j]$
(2) $T^2(C, j+1, k)$ with attributes $[2, 2, 1, 1]$ with the skipped token somewhere in the interior of span $[j + 1, k]$

For a simple choice of $f(\cdot)$ which is just the sum of its arguments, these functions fail optimality because we cannot arbitrarily choose any of the 4 ways to combine the solutions to B and C because choosing the first two solutions coalesce the subsequences in the middle resulting in this combination having lower $f(\cdot)$ than any of the other three combinations.

This shows that under this combination function we can run into situations where we cannot arbitrarily break ties.

The situation is even more dire, because by slightly changing our solutions to subproblems B and C and choosing a different $f(\cdot)$, we can create an example where the true optimal merge consists of taking suboptimal subtrees.

Say there are two solutions to $[B, i, j]$

(1) $T^1(B, i, j)$ with attributes $[5, 3, 1, 2]$ with all the skipped tokens forming a contiguous block on the right-boundary of $[i, j]$
(2) $T^2(B, i, j)$ with attributes $[5, 3, 1, 1]$ with the skipped tokens somewhere in the interior of span $[i, j]$

and say $[C, j + 1, k]$ has solution:

(1) $T^1(C, j + 1, k)$ with attributes $[2, 2, 1, 2]$ with all the skipped tokens forming a contiguous block on the left-boundary of $[i, j]$

Under $f(w, x, y, z) = w + x + 2 \cdot y + z$ solns (1) are the unique optima for both B and C, but when we choose those as our derivations to merge for $[A, i, k]$ we end up with a suboptimal solution with utility 20, due to coalescing the skipped subsequences in the middle. The optimal solution is to merge $T^2(B, i, j)$ and $T^1(C, j + 1, k)$ yielding utility 21.

4 Optimality of linear utility functions

Due to the thorniness of the combination function considered in the previous section, I will restrict attention to the combination function $g(X(A_1), ..., X(A_n)$ which simply performs vector addition over its inputs. I will call this combination function the additive combination function. For many different attributes (e.g. log probabilities, tokens counted, tokens skipped, constituents counted, etc.) this is a perfectly natural combination function.

I will first show that any function which takes a linear combination of its arguments $f(\cdot)$ satisfies optimality.

Theorem 1. *The additive combination function $g(\cdot)$ and any function which is linear in its arguments $f(\cdot)$ satisfy strong optimality.*

PROOF. Assume constituent B was derived via the following inference rule:

$$\text{RULE R}\ \frac{A_1 \quad ... \quad A_k}{B} \tag{5}$$

Each antecedent $A_1, ..., A_i, ..., A_k$ call the attribute vector of the j^{th} derivation of A_i $X^j(A_i)$.

We call it a greedy merge if B is obtained by choosing the best derivations of each antecedent - call their utility $f(A_i)$ for all $i \in [1, k]$.

We can write the attributes of the proof of B achieved in this manner as $X^{opt}(B) = g(X(A_1), ..., X(A_k), X(Rule\ R) = X(Rule\ R) + \sum X(A_i)$ where $X_{Rule\ R}$ is the inherent attributes of Rule R.

Clearly $f(X^{opt}(B) = f(X(Rule\ R) + f(\sum X(A_i)) = f(X_{RuleR}) + \Sigma f(A_i)$ by the linearity of $f(\cdot)$

This is the best value of $f(X_B)$ achievable for any proof of B using this inference rule because changing any of the proofs for any of the A_i would result in a lessened contribution to B's utility

More formally, switching the proof of A_d to any of its other proofs r would yield

$$f(X^{subopt}(B)) = f(X(Rule\ R)) + f(X(A_d)^r) + \sum_{i \neq d} f(X(A_i)) \tag{6}$$

$$= f(X(Rule\ R)) + f(X^r(A_d)) + \sum_{i \neq d} f(X(A_i)) \tag{7}$$

$$\leq f(X_{RuleR}) + \sum f(X(A_i)) \tag{8}$$

$$= f(X^{opt}(B)) \tag{9}$$

Where the inequality in 7 is due to $f(X(A_d)) \geq f(X^r(A_d))$. □

Thus as long as we use a linear utility function and the additive combination function, we can achieve optimality greedily with no additional constraints imposed on our attributes.

5 Optimality as a semiring

[3]'s work on parsing as deductive reasoning was generalized by [2] to show that may useful values for derivations in deductive parsing can be calculated via semirings. A semiring $\langle K, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$ is defined to be a set K with an addition operation $\oplus$ and multiplication operation $\otimes$ with appropriately defined identities $\mathbf{0}$ and $\mathbf{1}$ respectively, such that addition is associative and multiplication distributes over addition.

Goodman shows that a number of important parsing values can be obtained by framing them as semirings. For instance, Earley recognition can be considered parsing over the boolean semiring $\langle \{\text{TRUE, FALSE}\}, \vee, \wedge, \text{FALSE, TRUE} \rangle$. He shows a number of other useful values such as string probability: $\langle \mathbb{R}, +, \times, 0, 1 \rangle$.

The value of a node in an SPPF under the semiring formalism is:

$$\beta(v) = \oplus_{d \in I(v)} (k_d \otimes (\otimes_{u \in T(d)} \beta(u)) \tag{10}$$

Where d is a derivation in the set of derivations $I(v)$ for node v and u is an antecedent in the set $T(d)$ of antecedents of v under derivation d. k_d is the value associated with this particular derivation. For the string probability semiring, k_d would be the probability of the inference rule used in the derivation. For the boolean semiring it would be TRUE if there exists such an inference rule deriving v from its antecdents u and FALSE otherwise.

[4] gives an excellent presentation of the core deductive parsing algorithm over semirings and shows extensions of Goodman's work to the expectation semiring, the variance semiring, and further generalizations to gradients.

The interested reader should refer to both works cited above to understand the motivation behind semiring parsing as an analogue to deductive parsing.

We can represent utility value calculation with the additive combination function, as described above, as semiring parsing over the following semiring: $\langle (\mathbb{R}, \mathbb{R}^N), \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$.

Values in this semiring are tuples, the first element of which represents a real valued utility score and the second a real valued vector of attributes. Note in this semiring, k_d from equation 11 would represent the utility and attributes inherent to the derivation rule being triggered, the latter primarily referred to as X_{RuleR} in previous sections.

I will give definitions of addition, multiplication, and the identities. We will also see that the definition of a semiring as a set where multiplication distributes over addition will help us give a more precise definition of the utility functions $f(\cdot)$ which satisfy the optimality conditions presented in the previous section.

$$\begin{aligned} \langle u, \mathbf{a} \rangle \oplus \langle w, \mathbf{b} \rangle &\triangleq \langle u, \mathbf{a} \rangle \\ &s.t.\ \ u > w \\ \mathbf{0} &\triangleq \langle f(\theta_{inf}), \theta_{inf} \rangle \\ &s.t.\ \ \theta_{inf} \in \mathbb{R}^N \text{ is the infimum of} f(\cdot) \end{aligned}$$

We also define

$$\begin{aligned} \langle u, \mathbf{a} \rangle \otimes \langle w, \mathbf{b} \rangle &\triangleq \langle f(\mathbf{a} + \mathbf{b}), \mathbf{a} + \mathbf{b} \rangle \\ \mathbf{1} &\triangleq \langle f(\theta_0), \theta_0 \rangle \\ &s.t.\ \ \theta_0 \in \mathbb{R}^N \text{ is the all zeroes vector} \end{aligned}$$

Where "+" indicates vector addition.

Note that for all $\langle u, \mathbf{a} \rangle$, $u = f(\mathbf{a})$ where $f(\cdot)$ is the utility function of interest.

5.1 Distributivity of $\otimes$

We can derive a useful constraint on valid utility functions $f(\cdot)$ by using the requirement of the semiring that

multiplication distribute over addition:

$$\langle u_x, \mathbf{a_x} \rangle \otimes [\langle u_y, \mathbf{a_y} \rangle \oplus \langle u_z, \mathbf{a_z} \rangle] \tag{11}$$
$$= \langle f(\mathbf{c}), \mathbf{c} \rangle \tag{12}$$
$$\tag{13}$$

$$s.t.\ \ \mathbf{c} = \mathbf{a_x} + \mathbf{a_w} \text{ where } w = \begin{cases} y & u_y > u_z \\ z & otherwise \end{cases}$$

(12), by distributivity must be equivalent to (15):

$$(\langle u_x, \mathbf{a_x} \rangle \oplus \langle u_y, \mathbf{a_y} \rangle) \otimes (\langle u_x, \mathbf{a_x} \rangle \oplus \langle u_z, \mathbf{a_z} \rangle) \tag{15}$$
$$= \langle f(\mathbf{c}), \mathbf{c} \rangle \tag{16}$$
$$\tag{17}$$

$$s.t.\ \ \mathbf{c} = \mathbf{a_x} + \mathbf{a_w} \text{ where } w = \begin{cases} y & f(\mathbf{a_x} + \mathbf{a_y}) > f(\mathbf{a_x} + \mathbf{a_z}) \\ z & otherwise \end{cases}$$

Thus our $f(\cdot)$ defines a semiring over trees if and only if

$$f(\mathbf{a_x} + \mathbf{a_y}) > f(\mathbf{a_x} + \mathbf{a_z}) \iff f(\mathbf{a_y}) > f(\mathbf{a_z}) \tag{19}$$

Note that it is clear that if $f(\cdot)$ defines a semiring over trees then it satisfies optimality but it is unclear whether this is an iff.

Expanding elementwise under the definition $\mathbf{a_w} \triangleq [w_1, ..., w_N]$ we can rewrite (19) as:

$$f(x_1 + y_1, ..., x_N + y_N) > f(x_1 + z_1, ..., x_N + z_N) \tag{20}$$
$$\iff f(y_1, ..., y_N) > f(z_1, ..., z_N)$$

Note that this loosely corresponds to a notion of monotonicity, thought $\mathbb{R}^N$ is not ordered so monotonicity isn't well-defined

6 Preventing out-of-order processing

One of the most important requirements for valid utility calculations in our noise skipping paradigm is ensuring that given a derivation

$$\frac{[B, i, j] \quad [C, j, k]}{[A, i, k]} \ \mathrm{A} \rightarrow BC \tag{21}$$

that $[B, i, j]$ or $[C, j, k]$ are not, then, re-derived later yielding different utility values.

The issue posed by this out-of-order processing is that the utility value of $[A, i, k]$ is only computed when genuinely unique inference steps are triggered. This means if $[A, i, k]$ is derived once using a single inference rule and its utility calculated, and then one of its antecedents is re-derived with a new utility value, then the value for $[A, i, k]$ is not reset even though there exists a derivation of $[A, i, k]$ with improved utility (namely the one using the out-of-order derivation for its antecedent).

Note that there are four distinct ways an Earley state $[A, i, k]$ can be derived from unique inference steps:

(1)

$$\textsc{Rule1}\ \frac{[B, i, j] \quad [C, j, k]}{[A, i, k]}\ \mathrm{A} \rightarrow BC$$

(2)

$$\textsc{Rule1}\ \frac{[B, i, j] \quad [C, j', k]}{[A, i, k]}\ \mathrm{A} \rightarrow BC, j' \neq j$$

(3)

$$\textsc{Rule2}\ \frac{[D, i, j] \quad [C, j, k]}{[A, i, k]}\ \mathrm{A} \rightarrow DC$$

(4)

$$\textsc{Rule3}\ \frac{[D, i, j] \quad [E, j, k]}{[A, i, k]}\ \mathrm{A} \rightarrow DE$$

That is, $[A, i, k]$ can be generated by two identical rules but with different amounts of noise skipped (1 vs. 2) or via different rules consisting of the same last element (compare 1 & 2 to 3) or via completely different rules.

Typically,an Earley state stores information regarding what rule was triggered to generate it, what its start index is, and what its last index is. Thus unique Earley states would be generated for exs 1, 3, and 4 above, but identical Earley states would be generated for exs 1 and 2.

The conflict resolution scheme implicit in this work is that when a unique Earley state has been generated twice, we perform the requisite calculations to determine which derivation had the highest utility, and keep, associated with it, the attributes of the highest utility derivation. Thus, e.g., assuming all else is equal, we would keep the attributes from the derivation in 1 as opposed to those from 2 because 2 skips more tokens and is probably worse (by any reasonable utility function).

Something further important to note is that, though the identity of each unique $[A, i, k]$ state is tied to the rule used to generate it, each of those states is agnostic of the rules used to generate its antecedents. This is done for compression purposes. Otherwise each state would itself be an entire derivation tree, obviating the purpose of a shared-packed parse forest representation and enabling potentially $O(e^N)$ space complexity.

This compression of the history of antecedents is where the trouble of out-of-order processing can arise. What happens if we generate three states:

(1) $[A, i, k], A \rightarrow BC$, utility = 10
(2) $[A, i, k], A \rightarrow DC$, utility = 11
(3) $[A, i, k], A \rightarrow DE$, utility = 12

via the rules presented above, and then, $[C, j, k]$ is derived again with a new, best-derivation, that if incorporated into the already-processed states above would result in the following new states?

(1) $[A, i, k], A \rightarrow BC$, utility = 13
(2) $[A, i, k], A \rightarrow DC$, utility = 14
(3) $[A, i, k], A \rightarrow DE$, utility = 12

Because our processing is out-of-order the utility values are never updated and our Earley states would still

believe that $[A, i, k], A \rightarrow DE$, utility = 12 is the best derivation.

The problem then is that **out-of-order processing** results in a state (uniquely determined by its LHS, RHS, and span) having an incorrect utility score associated with it because it was completed and then one of its antecedents (one of the states used to derive it) is derived again later on resulting in a different utility score.

Since the Earley chart is organized such that states are stored in lists associated with their ending-index and we fill out the chart one index at a time, we can adjust the definition of **out-of-order processing** slightly, because rederivation could only ever happen to the rightmost antecedent. To see this note that $[C, j, k]$ and $[A, i, k]$ are stored in L_k but $[B, i, j]$ is stored in L_j. If we assume we have no empty productions or unary cycles (e.g. $A \rightarrow B, B \rightarrow A$), $j < k$ so we don't need to concern ourselves with $[B, i, j]$ getting rederived after $[A, i, k]$ has been completed.

Thus our working problem statement is:

Out-of-order processing occurs when a state $([A, i, k], A \rightarrow BC$, utility = $x)$ is created by a rule:

$$\text{RULE1}\ \frac{[B_1, i, i'] \quad \ldots [B_n, i'', j] \quad [C, j', k]}{[A, i, k]}\ \text{A} \rightarrow B_1 \ldots B_n C$$

and then at some later time, $[C, j', k]$ is processed again via a unique derivation to yield a state $[C, j', k]$ with new utility value such that rederiving

$$\text{RULE1}\ \frac{[B_1, i, i'] \quad \ldots [B_n, i'', j] \quad [C, j', k]}{[A, i, k]}\ \text{A} \rightarrow B_1 \ldots B_n C$$

ought to yield $([A, i, k], A \rightarrow BC$, utility = $y)$ where $y > x$.

6.1 How choice of data structure impacts out-of-order processing

Whether out-of-order processing occurs depends on what kind of data structure we use underlying our chart.

I will assume that Earley states are stored in abstract data types ADT_k indexed by their end-index and that each $i \in [0, |input|]$ is processed sequentially until all states in ADT_i have been processed, as is typical for Earley implementations.

Thus the key factor determining whether out-of-order processing occurs is the order in which states are processed within each ADT_k. I will now show that using a stack ADT or queue ADT produces out-of-order processing, while out-of-order processing can be avoided by using a priority queue sorted in order of decreasing start index with ties broken by a special topological sort over a small subset graph of the grammar.

6.1.1 Stack ADTs

Consider the simple grammar

Grammar 1

$$S := DC \tag{22}$$
$$C := AB \tag{23}$$
$$A := a \tag{24}$$
$$D := d \tag{25}$$
$$B := bbb \tag{26}$$
$$B := bb \tag{27}$$

applied to input "dabbb" there is one parse with no skipping (using rules 22-26) and one with a single token skipped (rules 22-25 and 27).

Crucially, both parses use the following inference rule to complete C:

$$(13)\ \frac{[A, 1, 2] \quad [B, 2, 5]}{[C, 1, 5]}\ \text{C} \rightarrow AB$$

Let's consider what happens when we process ADT_k with an underlying stack.

On the eve of processing ADT_5 it would likely look something like $[(b, 4, 5)]$.

After popping $(b, 4, 5)$ and then pushing states completed by that scan, we would have the following stack:

$[(B, [b, b, @], [2, 5])$
$(B, [b, b, @, b], [2, 5])$
$(B, [b, b, b, @], [2, 5])$
$(B, [b, b, @], [3, 5])]$

We would pop-off the top state (which has a single token skipped) and use it to complete C using the inference rule 23 above, yielding the a stack, where the $(B, [b, b, @], [2, 5])$ has been replaced by $(C, [A, B, @], [1, 5])$ which has a single token skipped.

We would then pop the top state again, completing S (with a single token skipped) via rule 12, replacing $(C, [A, B, @], [1, 5])$ with $(S, [D, C, @], [0, 5])$.

Next we would pop S off and store it as a completed sentential form.

We would then pop the next state $(B, [b, b, @, b], [2, 5])$, which would unsuccessfully attempt to scan past the end of the string.

We would then have the following stack

$(B, [b, b, b, @], [2, 5])$
$(B, [b, b, @], [3, 5])]$

Popping off the top state would derive C again yielding a new tokens-skipped value of 0.

The crucial issue here, is that the core logic of the Earley parser makes it so that when a state is derived which has identical production and span as one that already exists,

you don't add that state back to the ADT to be processed again. Therefore we would update the attribute values of $(C, [A, B, @], [1, 5])$ but we wouldn't push it to the stack to rederive $(S, [D, C, @], [0, 5])$, leaving $(S, [D, C, @], [0, 5])$ with permanently incorrect tokens-skipped value of 1.

Next we would pop $(B, [b, b, @], [3, 5])$ and use

$$(13)\ \frac{[A, 1, 2] \quad [B, 3, 5]}{[C, 1, 5]}\ \mathrm{C} \rightarrow AB$$

to derive $(C, [A, B, @], [1, 5])$ for a third time. Again, because the state already existed, we won't push back to the stack for further processing and our computation would terminate.

Note that in this case we had out-of-order processing resulting in our final sentential form to have an incorrect calculation of the best possible parse (in particular, the state $(S, [D, C, @], [0, 5])$ thinks the best derivation skips a single token). Thus there exist grammars and inputs where stack ADTs will result in out-of-order processing.

6.1.2 Priority queue ADTs

The proof of validity for certain priority queue ADTs is somewhat difficult.

As noted, out-of-order processing happens when a state is derived

$$\textsc{Rule1}\ \frac{[B_1, i, i'] \quad ...[B_n, i'', j] \quad [C, j', k]}{[A, i, k]}\ \mathrm{A} \rightarrow B_1 ... B_n C$$

and then at a later time its rightmost antecedent is rederived using the same rule it was derived with the first time.

This is the case we saw in the previous subsection. $(S, [D, C, @], [0, 5])$ was derived via rule 22, yielding a certain value for the best derivation of S, and then C was rederived later on using the same rule. Meaning that C was processed via rule 23 to yield S, S was processed and removed from the list and stored as a completed sentential form, then C was derived via rule 23 again yielding a new best derivation of S, but S's value was not updated because the state $(C, 1, 5)$ derived from rule 23 already existed in our Earley chart. (It is very important to note that a state $(C, 1, 5)$ is added to the chart anew iff it was derived via a different rule, since there was only a single rule deriving C, any time C got rederived would yield out-of-order processing)

We will show that if the Earley chart has ADT_k as a priority queue ranking states in decreasing order of their start index then we will get out-of-order processing if we break ties in the order the states were added in (FIFO) if we have unary productions. After showing that a simple priority queue ranked by start index with FIFO structure for each index fails, we will suggest a more robust method which is successful.

For a given derived symbol A, there are two cases, either A is derived from a rule with multiple terms on its RHS, or it is derived from a unary rule (note that we want to allow unary rules as long as they don't result in unary cycles!).

In the first case,

$$\textsc{Rule1}\ \frac{[B_1, i, i'] \quad ...[B_n, i'', j] \quad [C, j', k]}{[A, i, k]}\ \mathrm{A} \rightarrow B_1 ... B_n C$$

we know that the B_i are not empty and therefore $j' > i$. In which case, trivially, we will never have $[C, j', k]$ processed, then $[A, i, k]$ processed, then $[C, j', k]$ processed again, because all possible derivations of $[C, j', k]$ will be processed prior to $[A, i, k]$ because $j' > i$.

Therefore if we have no unary productions then this priority queue scheme works. Meaning, if we have no unary productions we can simply rank by start index and don't even need a tie-breaking scheme. If, however, we admit unary productions (even if they're not cyclic) we need a more advanced tie-breaking strategy to prevent **out-of-order processing**

Consider the second case

$$\textsc{Rule2}\ \frac{[C, i, k]}{[A, i, k]}\ \mathrm{A} \rightarrow C$$

The only way for out-of-order processing to occur is if $[C, i, k]$ is derived via some rule $\mathbb{N}$, it is then processed and used to derive A, and then its antecedent in rule $\mathbb{N}$ is rederived, as this would cause rederivation of $[C, i, k]$ after it was processed the first time. But since it was rederived via the same rule as last time, it wouldn't result in reprocessing and therefore the work performed in deriving $[A, i, k]$ wouldn't be overwritten. The important distinction to note here is that rederivation of $[C, i, k]$ is only a problem if it occurs via a rule that has been previously used to derive it. If it is derived via a new rule, then a unique state is created and $[C, i, k]$ is processed anew to rederive the updated value of $[A, i, k]$.

That is, our priority queue must have the property that if $[C, i, k]$ is derived by rule $\mathbb{N}$ with antecedent $[B_r, j', k]$, then the antecedent must never be placed on the queue again after $[C, i, k]$ is popped.

All of the $[B_r, j', k]$ for which $j' > i$ trivially precede $[C, i, k]$ because of the first ordering predicate of the priority queue.

For $[B_r, j', k]$ for which $j' = i$ we have that C is derived from $[B_r, j', k]$ via a rule

$$\mathrm{N}\ \frac{[B_r, i, k]}{[C, i, k]}\ \mathrm{C} \rightarrow B_r$$

We must show that it is possible to construct a grammar in which a $[B_r, i, k]$ derives $[C, i, k]$, $[C, i, k]$ is popped off of the queue and then $[B_r, i, k]$ is placed back on the queue. This would result in out-of-order processing showing we need a stronger tie-breaking scheme than simply FIFO ordering.

To show this, consider that all states completed on ADT_k have as their right corner terminal α_k.

We are going to expand our grammar into a directed graph where we have an edge from symbol B to A if there exists a rule $A \rightarrow \alpha B \rightarrow^* \alpha_i ... \alpha_k$ (with α potentially

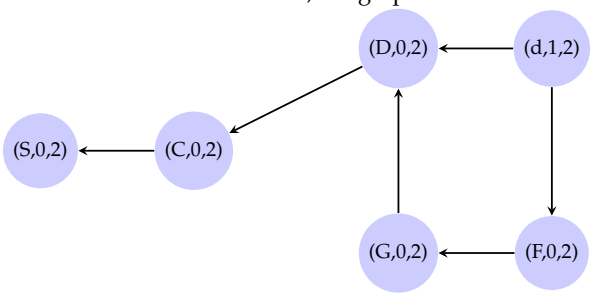
empty and * indicating transitive closure). Meaning, that A derives $\alpha_i ... \alpha_k$ via direct application of a rule with B as the rightmost term on the RHS.

Consider the grammar applied to input $d\ d$.

Grammar 2

$$S := C \quad (28)$$
$$C := D \quad (29)$$
$$D := d\ d \quad (30)$$
$$D := G \quad (31)$$
$$G := F \quad (32)$$
$$F := d\ d \quad (33)$$

This has associated with it, the graph



Note that because we have no unary cycles and because there are no empty productions a state can never derive itself. This means that the graph is always a DAG.

The states directly adjacent to (d,1,2) will be completed first, followed by the direct neighbors of those states in some arbitrary order.

For example, we add the first generation of completed states **[(D,0,2), (F,0,2)]**. We then process (D,0,2) adding (C,0,2) to get **[(F,0,2), (C,0,2)]**. Subsequent steps yield **[(C,0,2), (G,0,2)]**. We then fall prey to out-of-order processing because we process (C,0,2), then we derive (D,0,2) anew via (G,0,2).

Our ADT next becomes **[(G,0,2), (S,0,2)]** then **[(S,0,2), (D,0,2)]**, resulting in (S,0,2) being processed once, and then never processed again because the next time (C,0,2) is derived, it is once again via rule 29 and therefore no new state is created and the work done when (S,0,2) was derived the first time is not overwritten. Seen differently, (S,0,2) is reached first via the shortest path from (d,1,2). It is then never reached again because the long path to (C,0,2) updates (C,0,2)'s values, but because (C,0,2)-derived-from-(D,0,2) already occurred - that is, that edge has already been traversed, the algorithm terminates and doesn't notify (S,0,2) that its values have potentially been changed.

Because we add the states which could be completed this round in an arbitrary order, it is possible for there to be a path from the source (d,1,2) to C that is shorter than the alternative path from the source to C's antecedent (D,0,2), thus allowing derivation of C via rule 29 followed by another derivation of C via the exact same rule. This reapplication of the same rule via a different processing path causes (C,0,2) to derive (S,0,2) once, the first time it is derived via 29, but not update its value the second time it is derived via 29 several steps latter - preventing the original (S,0,2) from potentially being overwritten by the updated values of its antecedents.

6.1.3 Topological sorting

There is a very straightforward solution to this problem. Namely perform tie-breaking via topological sort. It is immediately apparent from the graph given above and from the properties of DAGs that in any topological sort, the antecedents of a given node C will always come before C itself, even if there exist short paths from the source to C.

Because of our 'no unary cycles' stipulation, there exists a global topological sorting of nonterminal symbols derivable by unary productions that is valid for all right-corners α_k. This means that, no matter what terminal completion begins the priority queue at index k there is a unique topological sorting which will tell us what order to perform the completions in thereafter. Thus we don't need to perform a new topological sorting every completion step - a single topological sort before parsing even begins will suffice! This can be shown via a proof by contradiction.

Theorem 2. *If there are no unary cycles in the grammar then we can construct a graph G in which each node is a nonterminal and there is a directed edge from symbol B to symbol A iff there is a unary production deriving A from B, such that:*

(1) *G is acyclic and has a valid topological ordering*
(2) *There exists some topological ordering over the symbols in G that is valid for all right corner symbols α_k.*
(3) *Any topological ordering of G is valid for all right corner symbols α_k.*

PROOF. Item one above is trivial, because, by definition there are no unary cycles in the grammar, therefore the graph of all unary derivations is acyclic.

For item two, assume that there exist two distinct symbols αj and α_k for $j < k$. Assume that at j there is a topological sorting for the states completable by $(\alpha_j, j, j+1)$. Call the corresponding graph G_j. Additionally, assume that there is a topological sorting for the states completable by α_k. Call the corresponding graph G_k. Assume by contradiction that the two topological sortings are inconsistent with one another, that is there exist two symbols $B \prec A$ in one sorting that are $A \prec B$ in the other sorting. Each G_j and G_k has a unique source, therefore any path through B to A in G_j and any path from A to B in G_k must pass through only nodes and edges which are contained in G. There being a path from B to A in one graph and another from A to B in another, would therefore make G cyclic, which by assumption, is not the case.

For item three, as noted above $G_j/(\alpha_j, j, j+1)$ is a subgraph of G. Therefore a topological ordering over G is also a valid topological ordering over $G_j/(\alpha_j, j, j+1)$. It is trivially true that $(\alpha_j, j, j+1)$ is topologically first in G_j since it

is the unique source, therefore the topological ordering of G_j is just the topological ordering of $G_j/(\alpha_j, j, j+1)$ with $(\alpha_j, j, j+1)$ inserted at the front. □

This proof shows that we can simply topologically sort the nonterminal symbols by unary production when preprocessing the grammar and use the value of each symbol in that topological sort as its value in the sorting predicate for each ADT_k.

6.1.4 Queue ADTs

I will omit the proof that standard queues experience out of order processing as it is a trivial extension of the same proof for priority queues without topological sorting.

References

[1] J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 13, no. 2, p. 94–102, 1970.

[2] J. Goodman, "Semiring parsing," *Comput. Linguist.*, vol. 25, pp. 573–605, Dec. 1999.

[3] S. M. Shieber, Y. Schabes, and F. Pereira, "Principles and implementation of deductive parsing," *J. Log. Program.*, vol. 24, pp. 3–36, 1995.

[4] Z. Li and J. Eisner, "First- and second-order expectation semirings with applications to minimum-risk training on translation forests," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1 - Volume 1*, EMNLP '09, (Stroudsburg, PA, USA), pp. 40–51, Association for Computational Linguistics, 2009.

Chapter 2: An algorithm for noise-skipping Earley parsing with tree-attribute based utility functions

Abstract

In the previous chapter, I characterized Earley parsing [1] with maximizing utility over tree attributes, as performing logical deduction over a semiring [2] [3] defined over a combination function $g(\cdot)$ and a utility function $f(\cdot)$. That formulation is useful because it provides *both* definite conditions under which a set of extractable attributes and a utility function over those attributes can be computed greedily *and* some minimal processing requirements to ensure that*correct* greedy calculations don't get incorrectly overwritten during the course of the parse.

That approach is deficient insofar as deductive logic isn't explicit about the core processing differences necessary to make the standard Earley algorithm capable of unifying states which have skipped portions of the input text and to track attributes of the optimal subtrees anchored at a given state.

This chapter will be self-contained. Anybody with an understanding of context free grammars but with no background in parsing will be able to learn the traditional Earley algorithm, the algorithm for extracting parse forests, and my innovations enabling noise-skipping.

First, I will briefly sketch the 3 primary steps of the Earley algorithm and the manner in which its data structures are populated to enable parse forest extraction [4]. Next, I will show the modifications necessary to the core algorithm in order to track parses over skipped portions of text and the additional data structures needed to ensure metadata for optimal parses can be maintained, such that efficient extraction of the parse forest is possible after parsing is done. Finally, I will discuss methods for calculating and maintaining four specific attributes of trees - tokens explained, constituents encountered, noise skipped, and tokens skipped - all of which are highly relevant to providing arguments for utility functions capable of giving ordering over the set of all trees.

Finally, I will discuss time and space complexity and give some speed benchmarks on example grammars. Discussion of algorithms to construct parse forests and extract trees from them will be given in subsequent chapters.

CONTENTS

Abstract 1
Contents 1
1 Introduction 1
2 Earley's algorithm 2
2.1 States 2
2.2 Charts 2
2.3 Scanning, Prediction, and Completion 2
2.3.1 Scanning 3
2.3.2 Prediction 3
2.3.3 Completion 3
2.4 Later formulations with parse forest metadata 3
3 Skipping noise 4
3.1 Noisy subspan metadata 4
3.2 Scanning and prediction 5
3.3 Completion 6
3.3.1 Attribute calculation 7
3.3.2 Completion metadata and the top K queue 8
4 Complexity 8
4.1 Time complexity 8
4.2 Space complexity 8
References 9
A Attribute and utility example 9

1 Introduction

In 1969, Jay Earley developed the first efficient algorithm to parse general context free grammar [1]. Previous algorithms had been developed for specific kinds of CFG and accomplished cubic or subcubic time on those special CFGs, [1] briefly covers some of those early developments. Earley's publication marked the first cubic time algorithm for general CFGs and marked a substantial development for both its excellent theoretical bounds and its fast practical performance. Earley's algorithm has been enduringly popular as well because it particularly well-suited to easy construction of parse forests [4] and because its flexibility with respect to the grammar's representation makes parsing with massive grammars practical with good filtering schemes [5] discusses a suite of algorithms which achieve excellent empirical speed-ups. I have also included as an appendix to this thesis, a chapter of original research I have conducted on filtering massive context free grammars for input to an Earley algorithm. Though noted in [5] and in the appendix, I will mention here that Earley's algorithm is particularly amenable to on-the-fly preprocessor filtering because it doesn't rely on costly a priori computation of LR tables or shift/reduce tables like other competitive fast algorithms like GLR and its offshoots. I will note that an important addendum to the work in [5] is that only certain kinds of filtering are suitable for noise-skipping parsing. The algorithm I present in the appendix is suitable for the purpose, but the algorithms presented in [5] which require pre-computing left and right corners present in the input because we allow parses to complete despite skipping over arbitrary amounts of text making detecting the presence of left and right corner in the input text ill-defined. Chapters 5-7 of [6] provide excellent descriptions of the properties of

context free grammars and definitions related to parsing, derivations, and trees.

2 Earley's algorithm

2.1 States

Earley's algorithm was originally presented and is most succinctly described in [1]. An excellent exposition can be found chapter 30 of [7] as well as source code for an object oriented implementation in Java. [7] strays very little from Earley's original notation and concepts so I will use their presentation as a basis for the forthcoming discussion - this presentation will be somewhat different than others in that it will be highly procedural as opposed to the declarative style popular in more "deductive" approaches. The notation is also an intuitive extension of that used in the deductive logic presentation I gave in the last chapter.

Earley's algorithm is part of a general family of parsers known as chart parsers. The atomic unit of information used in Earley's algorithm is the *state*. A state can be thought of as representing both a summary of information confirmed thus far (up to index j of the string, e.g.) and also a hypothesis of information in the string which has yet to be processed. Example states are given below:

(1) (root, [@, Sentence], [1, 1])
(2) (sidenetwork, [C, @, sidenetwork, D], [1, 2])
(3) (sidenetwork, [C, sidenetwork, D, @], [1, 6])

A state is a 3-tuple whose leftmost member is a constituent, whose middle element is an array of vocabulary items representing a rule that exists in the grammar, as well as a dot indicating how much of the rule has been confirmed so far, and finally two numbers $[i, j]$ which indicate the beginning index of the span of input string covered by the rule and the (exclusive) upper bound of the interval consumed so far - it can also be thought of as the index in the string where the @ symbol sits. This is an abuse of notation - we should really be using $[i, j)$ notation as it means only tokens i through j-1 are explained by the state but in order to avoid adjacent)s I have opted to use square brackets for both the lower and upper bound.

For example, (1) represents a prediction that the rule $root \rightarrow Sentence$ will be completed. The @ indicates that no tokens have been consumed to confirm that hypothesis and the index tuple $[1, 1]$ indicates that the hypothesis says index 1 will be the starting place of the constituent called *Sentence* and that index 1 is the location of the last-read token (in this case no tokens have been read in so far so the end and start indices are the same).

(2) tells us that we are predicting a constituent of type sidenetwork will be completed at start index 1 via the rule $sidenetwork \rightarrow C\ sidenetwork\ D$. Furthermore, the location of the @ tells us that the terminal symbol C has already been read in, and the end index $j = 2$ tells us that that C was located at index 1 in the input string and that the the dot is now located at index 2 of the input string. Such a state may be incremented by finding that some arbitrary span beginning at index 2 can be derived by a subtree whose root is $sidenetwork$.

Finally, state 3 is an example of a state. It tells us that the rule $sidenetwork \rightarrow C\ sidenetwork\ D$ matched the text starting with the token C at index 1 and a D at index 5 (as noted 6 is the location of the dot, which is at the end of the rule, forming an exclusive upper bound for the interval of text matched by the rule). In the traditional Earley context, this means that the middle $sidenetwork$, via some derivation (or potentially multiple), began at index 2 and matched every token through index 4. We will find that in the noise-skipping context it may be the case that $sidenetwork$ matched $[2, 5]$ but skipped the middle token (3) or that it only matched some subset of the interval $[2, 5]$ and that there was some skipping between the end of $sidenetwork$'s interval and the D consumed at index 5 to complete the state.

2.2 Charts

States are organized into a list of charts where each index j in the chart represents a location in the input string and stores all of the states whose end index is j. Thus, because end indices of states represent exclusive upper bounds, the list of charts will always be longer than the input string by 1.

A chart can be built out of pretty much any dynamically sized data structure and can either maintain all states ever encountered whose end index is j, or can be added to and modified in-situ such that a state is popped off (dequeued, polled, etc.) as it is processed. For efficiency purposes, the chart will typically contain a hash table mapping symbols in the vocabulary to a set of states in the chart who have that symbol immediately after their dot. Furthermore, for efficient processing, a chart will keep track of every state that has ever been added to it so that if a state gets added once (due to some derivation) and then added again (via a new, different derivation) it is not added to the chart to be processed once more, but it is merged with the previous version encountered in a way parsimonious with the style of parsing. In [4] this is done by merging the reduction and predecessor lists. In our noise-skipping variant we will also merge attributes for the subtrees derived by each state.

What is meant by a state being identical to another state previously added can depend on the context, but in the context of this research states will be identified by their LHS, their rule and dot location, and their indices (i.e. states with the same string form as given above are "identical").

Though in standard Earley parsing it is irrelevant what data structure you use to underly each chart. I showed in the previous chapter that parsing correctly over the semiring with greedy utility evaluation requires a data structure with states sorted in order of start index i and with ties broken via a topological sort over the grammar.

2.3 Scanning, Prediction, and Completion

The input sentence is processed one index at a time starting by populating chart one with a set of starting states (I will only consider a single unique starting state (root, [@, Sentence], [1, 1])). Each state can have exactly one of three unique operations performed on it, which

will either add new states to the chart, result in the completion of a "full" interpretation (e.g. we found a span which constitutes a full sentential form for the grammar), or fail to do either.

2.3.1 Scanning

Scanning is performed while processing states of the form St ≡ (sidenetwork, [C, sidenetwork, @, D], [1, 5]) where the next item after the dot is a terminal part of the vocabulary. Scanning St checks whether the element of the string at index 5 matches the element after the dot, namely *D*. If it matches then St spawns a new state St' ≡ (D, [D, @], [5, 6]) and puts it on chart $j + 1$. Scanning is the simplest operation as it requires a simple check to see if the next element of the input matches the element expected after the dot, and if it does creates a state indicating consumption of the terminal *D* beginning at *j* and located on chart $j + 1$, or it fails to find a token matching the element following the dot and the algorithm moves on to process the next state in the current chart (or, if there are no more states in chart *j*, move on to chart $j + 1$).

2.3.2 Prediction

Prediction is the nonterminal analogue of scanning. It is performed while processing states of the form St ≡ (sidenetwork, [C, @, sidenetwork, D], [1, 2]) where the element immediately following the dot is a nonterminal part of the grammar. A state such as St can be intuitively thought of as hypothesizing that a constituent of type *sidenetwork* will begin at index 2. In order to perform this processing, we must find every rule in the grammar containing *sidenetwork* as its LHS and initialize a state in chart $j + 1$ that hypothesizes the completion of that rule for each rule in the grammar beginning with the element following the dot. For example if our grammar contained the rules $sidenetwork \rightarrow C\ sidenetwork\ D$ and $sidenetwork \rightarrow C\ D$ then two new states would be initialized on the following chart:

St' ≡ (sidenetwork, [@, C, sidenetwork, D], [2, 2])
and
St'' ≡ (sidenetwork, [@, C, D], [2, 2])

Note that so far the two operations discussed create new states unrelated to the state that spawned them, but they don't actually create successor states to St - that is, they don't move the dot in St, indicating that more of our hypotheses have been confirmed as more the text has been consumed (there are some implementations of scanning which have scanning do so by just moving the dot in St and not creating a new St' at $j + 1$) - the job of creating successor states is the *completion* step.

2.3.3 Completion

Completion is performed on any state whose dot occupies the final position in the array of elements. For example St ≡ (D, [D, @], [5, 6]) would be a candidate for completion because @ occupies its final position. Completion in chart *j* occurs by searching chart $j-1$ (efficiently if a hash map from symbols to states has been implemented) for states whose element immediately following the dot matches with the LHS of St. For example, completion of St may search chart $j - 1$ and find states

St' ≡ (sidenetwork, [C, sidenetwork, @, D], [1, 5]) and St'' ≡ (sidenetwork, [C, @, D], [4, 5])

and produces successor states for St' and St''

St'_{new} ≡ (sidenetwork, [C, sidenetwork, D, @], [1, 6]) and St''_{new} ≡ (sidenetwork, [C, D, @], [4, 6]) Being that the successor states are pushed onto the chart currently being processed, they can be used in the current iteration to complete other states which in turn may be eligible for prediction, completion, or scanning.

By the conventions used here, the algorithm is said to have succeeded when a sentential form is completed.

2.4 Later formulations with parse forest metadata

The astute reader may have noticed that this algorithm doesn't constitute a parsing algorithm but rather a *recognition* algorithm. This is so because the algorithm reports whether there are any completed states corresponding to a sentential form, but it doesn't actually stipulate a way to extract the trees corresponding those to states. Earley gestures towards a simple algorithm that converts the recognizer into a parser [1] but it was discovered by Tomita in 1986 that Earley's algorithm for parse extraction will generate spurious parses [8]. Scott [4] reports that other spurious attempts were made over time, including that in the reference textbook implementation cited previously [7], and shows a provably correct algorithm for building a parse forest data structure known as a binarized Shared-Packed Parse Forest which succeeds in producing a cubic-sized SPPF in cubic time which is capable of storing an exponential number of trees. Though she provides no algorithm for extracting the trees in any particular order. Though other works have been dedicated to performing operations and modifications to eliminate ambiguities in the parse forest (this is used gainfully to simplify expression grammars and a language spec similar to that of the Java language specification) [9], to my knowledge the next chapter of this thesis represents the first time anybody has developed a method for extracting trees from the parse forest in a ranked-order in polynomial time.

I will omit Scott's presentation of the Earley algorithm as it has a much more declarative flavor and relies on different vocabulary and notation to describe it. Instead I will give the impact its processing has on the Earley algorithm given so far. Scott's core contribution to the Earley algorithm is the addition of data structures to track the derivational relationships between states that she dubs "predecessor" and "reduction".

Scott's work also presents an algorithm to construct the forest itself from the "predecessor/reduction" data structures but I will defer discussion of those algorithms to the next chapter.

Scott maintains two mappings from indices to lists of states, one labeled predictions and the other labeled reductions. She states that during what I refer to here as the completion phase, for every state with LHS a terminal member of the grammar $terminal \equiv (a_j, [a_j, @], [j, j + 1])$ and state completed as a consequence $q \equiv (A, [\alpha, @, a_j, \beta], [i, j])$

and new successor state $p \equiv (A, [\alpha, a_j, @, \beta], [i, j{+}1])$ a mapping is created in q's predecessor map with key j pointing to a predecessor list containing p, indicating an edge labeled j from q to p. I have found that this is incorrect and instead the mapping must be in p's predecessor map indicating an edge from p to q with label j. If you read [4] you will find my indexing scheme is different, where she refers to i as j and refers to $i + 1$ as j. I have found that my indexing scheme is much more intuitive. Note that this mapping is only made if α is a non empty list of elements in the grammar. That is, the dot index for q must not be equal to 0.

I have found that the rest of her predecessor/reduction construction generates correct results. During the completion phase of a state whose LHS is a nonterminal, $nt \equiv (B, [\beta, @], [j, k])$ find all states affected by the completion of beta, $q \equiv (D, [\delta, @, B, \mu], [i, j])$ and produce new state $p \equiv (D, [\delta, B, @, \mu], [i, k])$ and create a reduction edge labeled j from p to nt. If δ was non-empty, then also add a predecessor edge from p to q labeled j. Then, per the usual completion calculus we add p to the chart at index k.

Thus the general relationship is, if a state p is a successor of q (same state but with the dot moved) and q's dot was not at the start position, label a predecessor edge from p to q with label j where j was the end index of q. If p is being produced via a state nt with nonterminal LHS then create a reduction edge from p to nt with label j where j is both the start of nt and the end of q.

Additionally, note that if at any point a state is created which is "identical" to another state per the definition in section 1 then you simply merge the predecessor

3 Skipping noise

While the previous chapter gave a theoretical introduction to the parsing as deduction and ranking trees as an operation over semirings, the last two sections of this chapter have laid out important practical groundwork explaining the context in which the following few sections and the remaining chapters of the thesis are situated as well as concrete guidelines for implementation. I will now present a definition of the noise-skipping problem and some motivating applications as well as make a distinction between skipped "noise" and skipped "tokens". The remaining subsections will concern themselves with the modifications necessary to generalize the implementation described in the last two sections to noise-skipping and the acquisition of "attributes" of the best subtree derivable from a given state.

Noise skipping parsing is a form of parsing that allows completion of rules to occur discontinuously. Whereas before we required that states and their successors be directly aligned with one another, during noise skipping we allow some number of symbols to intervene. A simple example is parsing the grammar $S \rightarrow A\ B\ C$. Whereas before we could only parse the literal string "A B C" a noise skipping parser can parse "A d B C", "A d d d d B d d C" and even "A B C C" where we recognize the parse which skips the first C in favor of the second C as constituting a valid parse. This discontinuity modifies the behavior of all three of the operations in the Earley algorithm and also requires some additional overhead to maintain efficiency. Furthermore, generalizing Earley parsing to noisy data naturally leads us to another important question, addressed theoretically in the previous chapter, of all the possible parses (of which many are induced due to relaxing the constraints to permit noise) how do we find the best ones, and how do we maintain attributes describing the best ones?

Though other approaches have concerned themselves with parsing noisy data, e.g. [10] which does so for the GLR algorithm, to my knowledge this thesis is the first to unify the Earley algorithm, noise skipping, SPPF construction, and in-order tree extraction ranked by attribute-based utility functions into a single approach. Furthermore, because this is a modification of the Earley algorithm it both accomplishes its goal via a relatively simple extension of the original algorithms (compare this to the practical difficulties of implementing the GLR* algorithm in [10]) and reaps the benefits of efficient grammar filtering which is explored further in the appendix of this thesis and in [5] in the non noise-skipping context.

One of the most important applications of the noise skipping approach is extracting information regarding some substructure of a text found meaningful for a given purpose while ignoring the rest of the text. In this case, it is important to distinguish achieving parses which skip tokens outside the vocabulary of the grammar, which I refer to as **noise skipping**, and permitting parses which skip valid elements of the grammar (e.g. the parse "A B C C" given earlier), the latter of which I refer to as **token skipping**. The reason for this distinction is twofold: first, it is possible to skip noise with only O(n) space and time overhead during preprocessing and second, when comparing two different parses by their utilities it is typically meaningful to draw a distinction between the number of tokens skipped and the amount of noise skipped. That is, one may consider it more egregious to skip a token than it is to skip noise as noise is entirely unexplainable by the grammar whereas a token skipped may, under some interpretation, be explainable in context by the grammar. Thus, I generically refer to noise-skipping as the task of parsing which permits skipping of any parts of the text, though I also reserve the specific technical dichotomy laid out above between noise-skipping and token-skipping.

It is important to note that is possible to permit an arbitrary amount of noise skipping or to parametrize it with some maximal bandwidth of skipped noise we'd like to permit between spans of explained tokens. There is little practical difference between the two approaches so I will defer elaboration on this point until the complexity analysis in section 4.

3.1 Noisy subspan metadata

Though skipping noise comes essentially for free (the O(n) preprocessing scan is likely necessary anyway if one is using a part of speech tagger, named entity recognizer, etc.), it comes at the cost of constructing and maintaining

at the outset two data structures as well as a third data structure that plays an important role in caching computed data during the parse so that it doesn't need to be recomputed later on.

During the preprocessing step we maintain a map from indices to indices of the form $[i, j]$ which indicate the beginnings and ends of spans of noise. That is, an entry $[i, j]$ indicates that there are unexplainable noise tokens at indices i through $j - 1$. This makes forward scans efficient and will play a prominent role in scanning and prediction because it allows us to skip over entire sections of noise at once. We must also maintain and inverted map of $[j, i]$ mappings ends of noisy spans to the beginning. This will play an important role during completions so that we can skip directly over entire sections of noise at once as opposed to checking each index that has noise. These two maps greatly impact performance when the input text has highly concentrated noise and large amounts of it.

We also maintain a third data structure that stores the amount of noise in a given span. Thus it is a map from integer tuples $[i, j]$ to integer counts of the number k of noise tokens in that range. This is useful any time we perform a calculation indicating a state can skip over some range $[i, j]$. It is important, in order to maintain correct attributes, that we know what portion of $[i, j]$ was tokens and what portion was noise. Caching this information is motivated by the belief that if a given skipped span has the potential to be used in one completion, it is likely to be used multiple other times, and it saves us the overhead of recalculating. At a minimum, even if multiple derivations don't use the same skipped span, we are guaranteed that for a given completion step which uses it, it also had to have been used for a prediction or a scan earlier on, meaning that at the very least, caching that information saves us one scan through the range to count the number of tokens. The next several sections will make explicit how these data structures, and other modifications, are used during processing to achieve fast, correct results. I would like to note that it is possible to achieve greater time efficiency using dynamic programming if you are interested in computing the noise between all $O(n^2)$ spans but in practice not all of the spans end up being needed during the parse, and in fact it's more typical to see only $O(n)$ spans being needed or $O(w \cdot n)$ where w is the max number of tokens allowed to be skipped between two explained tokens in any given parse. Thus it is of greater practical benefit to calculate the spans which come up during processing and cache their results to be used later if needed.

A crucial intuition into why preprocessing noise is useful is that it takes a single glance at a noise token to know it will never be a part of any parse at all. When it comes to skipping tokens, we need to consider all possible hypothesis crossing its index to see which, if any, do not use it. Thus precomputing the noisy spans allows us to avoid the expensive computation requisite for skipping regular tokens.

3.2 Scanning and prediction

Scanning and prediction are almost entirely identical to their standard Earley parsing variants except for three main differences. Recall from section one that scanning occurs by taking states $s \equiv (A, [\alpha@, a, \beta], [i, j])$ and checking to see if the j^{th} element of the input text a_j matches a. Recall additionally that prediction occurs by taking states $s \equiv (A, [\alpha@, B, \beta], [i, j])$ and finding all rules $B \rightarrow ...$ and predicting a state of the form $s' \equiv (B, [@, ...], [j, j])$. Both methods are unified by the notion that they take the next symbol following the dot and check to see if its possible to find a substring starting at j which matches that symbol. Noise skipping differs insofar as doesn't require the following substring to begin at j. Instead we calculate a list of valid indices $j'_1, ...$, tabulating the number of tokens/noise skipped in the range from j to each valid index and then perform the prediction/scan at each index j'. A naive approach would be fairly trivial, but in order to exploit the precomputation done in the previous step I propose a somewhat more sophisticated algorithm given below This algorithm doesn't allow a state to begin

```
1:  procedure GETPREDICTEDINDICES(s, j)
2:      s ← state being processed
3:      j ← end index of s
4:      noisySpans ← precomputed
5:      curr ← j
6:      noiseSkipped ← 0
7:      w ← Maximum allowable skipped tokens
8:      indexToNoiseSkipped ← [(j, 0)]
9:      if s.DotIndex = 0 then
10:         return indexToNoiseSkipped
11:     else if s.DotIndex > 0 then
12:         noiseSkipped ← noisySpans.get(j) + 1 − curr
13:         curr ← noisySpans.get(j) + 1
14:     for i = 1; i <= w ∧ i < len(sentence); i + + do
15:         curr ← curr + 1
16:         if curr ∈ noisySpans.keys() then
17:             noiseSkipped ← noisySpans.get(j) + 1 − curr
18:             curr ← noisySpans.get(j) + 1
19:         if curr < len(sentence) then
20:             indexToNoiseSkipped.append((curr, noiseSkipped))
        return indexToNoiseSkipped
```

with skipping (this is captured by checking the dot index) as that would violate an invariant maintained by the algorithm that the start and end indices of a state indicate the locations of the first and last tokens explained in that span. Furthermore, this algorithm only returns indices which have a token at them and it directly calculates up to w such indices, in time proportional to w plus the number of spans of noise. This is especially time efficient if w is small (say 3) but there is a single large span of noise (say of length 100). This would mean that we carry out on the order of 4 calculations, as opposed to on the order of 103. This method is not very helpful if the noise is incredibly fragmented (e.g. each noisy span is ≈ 1 in size).

Thus, under my noise-skipping variant, scanning and prediction occur via calling the routine above, creating new states at each index j' in the usual Earley sense, and caching the amount of noise skipped in range $[j, j']$.

Scanning and prediction require two additional modifications related to additional metadata maintained for each state. Due to the requirements of preventing out-of-order processing discussed in the previous chapter, each new state is assigned a topological sort number to aid in the calculation of the priority queue underlying each chart. The topological sort number is determined by the topological sort number of the LHS. Additionally, each state maintains a count of the total amount of noise and tokens skipped, so the new state generated in a given scan or prediction, e.g. $s' \equiv (B, [@, ...], [j', j'])$ would store $j' - j$ as an additional piece of metadata. Additionally, its important to note that each new state is put onto chart j' in the case of prediction and $j' + 1$ in the case of scanning (because scanning produces states $s' \equiv (a_{j'}, [a_{j'}, @], [j', j' + 1])$).

3.3 Completion

Completion requires more complex modifications, because the noise-skipping domain changes which states are completable by a given state undergoing completion, furthermore it is the stage at which attributes are calculated and associated with new states. I will first discuss modifications needed to determine which states are completable, how we create the resulting successor states and predecessor/reduction relations. In the next section I will discuss four specific attributes and then following that I will discuss data structures requisite for efficient extraction from the SPPF forest - a topic that will discussed in detail in the next two chapters.

As mentioned previously, completion occurs when a state $c \equiv [\Gamma, []\gamma, @], [j', k])$ is processed. Applying the completion process to c searches for all states of the form $q \equiv (D, [\alpha, @, \Gamma, \beta], [i, j])$ whose symbol after the dot matches the LHS of state c. From q, the new successor state $p \equiv (D, [\alpha, \Gamma, @, \beta], [i, k])$ is produced. In the context discussed earlier, completions were only valid if $j = j'$ and searching for completions would occur only by checking chart j' for states whose symbol after the dot match c's LHS.

The first major procedural change to the completion algorithm is the introduction of a new method to search for candidates q which could be completed to produce p. As in the previous subsection, it would be possible to simply search every index of the chart going back to some specified time horizon (or back to the beginning of the input) and search the charts by the symbol after the dot in each state (note that I mentioned earlier the chart typically has a data structure mapping symbols to states whose symbol after the dot match that symbol). Instead, I will show a more sophisticated algorithm which exploits the precomputed noisy spans in the reverse direction, in much the same way as the GetPredictedIndices(s,j) did so for the forward direction. The algorithm consists of a wrapper method which extracts states from each chart by the symbol after their dot and a utility method which does the reverse scan described above for getting the indices which could contribute to a valid completion. Get-

```
1:  procedure GETCOMPLETED(s, lhs)
2:      s ← state undergoing completion
3:      lhs ← the lefthand side of s
4:      completedStates ← {}
5:      charts ← the array of charts for the parse
6:      validIndices ← GetValidIndices(s)
7:      for index ∈ validIndices do
8:          completedStates.append(chart[index].getByAfterDot(lhs))
        return completedStates

    procedure GETVALIDINDICES(s, i)
2:      s ← state being processed
        i ← start index of s
4:      curr ← j
        noiseySpansReverse ← precomputed
6:      w ← Maximum allowable skipped tokens
        indices ← [start]
8:      for i = 1; i <= w ∧ i < len(sentence); i + + do
            curr ← curr − 1
10:         if curr ∈ noisySpansReverse.keys() then
                noiseSkipped ← noisySpans.get(j) + 1 − curr
12:             curr ← noisySpans.get(j)
            if curr < 0 then
14:             return indices
            indices.append(curr)
        return indices
```

ValidIndices(s,i) differs from GetPredictedIndices(s,j) primarily because it works in reverse and because it doesn't calculate noise skipped between each range. This is so because for any given completion that ends up being used, we are guaranteed to have already computed the noise skipped in the range of that completion during a forward sweep in the prediction or scanning phase.

The two other important difference to the core algorithm is the labeling conventions for the reduction and predecessor edges. As mentioned previously, the generalization is that when a state processing completion for a state nt, if p is a successor of q, where q was completed by nt, (i.e. p and q are the same state but with the dot moved) and q's dot was not at the start position, label a predecessor edge from p to q with label j where j was the end index of q. If p is being due to completion of state nt with nonterminal LHS then create a reduction edge from p to nt with label j where j is both the start of nt and the end of q.

The modification relevant to noise-skipping is that the start index of nt is not necessarily j but could be some $j' \geq j$. There are two cases: if nt had a terminal as its LHS then we still use j as the label for the predecessor edge from p to q, where j is the end index of q. If, however, nt had a nonterminal as its LHS then instead of using j, the end index of q, we use $j' \geq j$, the start index of nt as the label for all predecessor *and* reduction edges from p to q. It is a bit curious that the nonterminal case

and terminal case behave differently, namely that the terminal case, just as in the standard Earley algorithm, still uses q's end index j to label predecessor edges while the nonterminal case pivots and uses j' to label both edge types. The reason for this is subtle and has to do with the way terminal and nonterminal nodes are considered in the SPPF construction algorithm developed by Scott [4]. When I give the standard formulation of Scott's algorithm in the next chapter as well as my extensions necessary for noise-skipping it will become clear why this labeling scheme is correct.

An important note to reiterate from earlier, is that in the utility function-enabled setting we are discussing that every new state produced must be given a topological sort value per the discussion in the previous chapter to prevent out-of-order processing.s

3.3.1 Attribute calculation

As shown in the previous chapter, well-behaved attributes can be maintained greedily (calculated at the time of new state construction and never need to be recomputed) so long as the utility function used to maximize over the attributes meets the optimality criteria given in the previous chapter. Though the previous chapter primarily discussed attributes as being associated with constituents, they can also be thought of as connected to states, or in the parlance of the last chapter, to items of deductive logic (axioms, and theorems). I will preface the forthcoming concrete discussion, with a theoretical note on utility functions $f(\cdot)$ attribute vectors $X(S)$, where S is a state. In particular, in order to ensure the correctness of the greedy algorithm, we need to ensure topological sort information is maintained upon creation of each new state and we need to ensure that when a state S' is produced which "identical" to one that has been produced earlier S, that we set the attributes of the merged state S'' to be those of the state $argmax_{s \in \{S, S'\}}(f \circ X)(s)$

I will discuss 4 attributes which meet the criteria in the previous chapter and which I have found particularly useful for describing the utility of parse trees within the context of noise-skipping parsing: number of constituents in the tree, number of tokens skipped, amount of noise skipped, and tokens explained (where here I use the token vs noise dichotomy given in the beginning of section 3). It is possible to explain the methodology for calculating attributes via an inductive logic with some special set of inference rules and axioms for each attribute type, but I will disprefer that formulation in favor of the procedural explanation given below:

(1) **Tokens explained** During the scanning phase, if a state $p \equiv (a_i, [a_i, @], [i, i+1])$ is produced to denote that a terminal has been consumed, that state is initialized to have its TokensExplained counter set to 1.
For newly created states p constructed during the completion phase, there are two cases for how TokensExplained is tracked. If state $nt \equiv (B, [\beta, @], [j', k])$ is undergoing completion , where B is a nonterminal and is producing new state $p \equiv (D, [\delta, B, @, \mu], [i, k])$ as a successor to some state $q \equiv (D, [\delta, @, B, \mu], [i, j])$ then, if δ is non empty, then $p.TokensExplained$ is initialized to $q.TokensExplained$. Otherwise it is initialized to 0. After the initialization, $p.TokensExplained$ is then incremented by $nt.TokensExplained$.
In the case that $nt \equiv (\alpha_{j'}, [\alpha_{j'}, @], [j', j'+1])$ has a terminal as its LHS, then we always set $p.TokensExplained$ to $q.TokensExplained$+$nt.TokensExplained$. Note, however, that per the update in the scanning phase, $nt.TokensExplained$ will always be 1.

(2) **Constituent count** ConstituentCount is only updated during the completion phase. If state $nt \equiv (B, [\beta, @], [j', k])$ is undergoing completion, where B is a nonterminal and is producing new state $p \equiv (D, [\delta, B, @, \mu], [i, k])$ as a successor to some state $q \equiv (D, [\delta, @, B, \mu], [i, j])$ then, if δ is non empty, then $p.ConstituentCount$ is initialized to $q.ConstituentCount$. Otherwise it is initialized to 0. After the initialization, $p.TokensExplained$ is then incremented by $nt.ConstituentCount$. If μ is empty, then p has now been completed and we increment $p.ConstituentCount$ by 1.
In the case that $nt \equiv (\alpha_{j'}, [\alpha_{j'}, @], [j', j'+1])$ has a terminal as its LHS, then we always initialize $p.ConstituentCount$ to $q.ConstituentCount$ and then increment it by 1 if μ is empty.

(3) **Noise skipped** NoiseSkipped for a newly produced state can be efficiently calculated at completion time by leveraging the information cached in the noise skipped in range metadata computed during scanning and prediction. Again there are two cases: If state $nt \equiv (B, [\beta, @], [j', k])$ is undergoing completion, where B is a nonterminal and is producing new state $p \equiv (D, [\delta, B, @, \mu], [i, k])$ as a successor to some state $q \equiv (D, [\delta, @, B, \mu], [i, j])$ then, we first extract the cached noise skipped from $[j, j']$, call it $NoiseSkippedFrom([j, j'])$. If δ is non-empty then we initialize $p.NoiseSkipped = q.NoiseSkipped + NoiseSkippedFrom([j, j'])$ we then increment $p.NoiseSkipped$ by $nt.NoiseSkipped$.
In the case that $nt \equiv (\alpha_{j'}, [\alpha_{j'}, @], [j', j'+1])$ has a terminal as its LHS, then we extract $NoiseSkippedFrom([j, j'])$ as normal and simply set $p.NoiseSkipped = q.NoiseSkipped + NoiseSkippedFrom([j, j'])$.

(4) **Tokens skipped** TokensSkipped is calculable in a manner similar to NoiseSkipped. We calculate a value inherent to the range $[j, j']$ called $TokensSkippedFrom[j, j']$. It is defined as $j' - j - NoiseSkippedFrom([j, j'])$ (note this is just the size of the range skipped and the amount of noise in it). Once $TokensSkippedFrom[j, j']$ is calculated, the method of calculating $p.TokensSkipped$ is identical to that given for $p.NoiseSkipped$ except with $x.TokensSkipped$ substituted for all occurrences of $x.NoiseSkipped$.

I will include, in the appendix of this chapter, an example grammar with example utility function for which these attributes can meaningfully discern between optimal parses.

3.3.2 Completion metadata and the top K queue

In order to enable easy construction of the utility function-optimized SPPF and fast extraction of the top parse trees evaluated over the utility function, we need to maintain two more data structures during run-time.

In order to build the SPPF we will a map of **completion metadata**, referred to here, as *completions*, which maps tuples (nt, q) to the vector of attributes $X(p)$ which is completed by combining nt and q, note that while p is uniquely determined by (nt, q), the relationship is not bijective, multiple tuples (nt', q') could be used to produce p.

I claim that we only need to add the mapping $(nt, q) \mapsto X(p)$ in the *completions* map the first time it is encountered, and never need to update it again in order to maintain the most optimal set of attributes associated with deriving p from (nt, q). This is so because if the algorithm adheres to the topological sorting criteria given in the previous chapter then nt and q are never rederived once p is derived for the first time. That means that we know exactly what the best derivations of nt and q are by the time we reach p. Therefore if p is rederived later on it can only be via a different pair (nt', q'), because rederiving it with the same (nt, q) would constitute out-of-order processing. Thus, since we know our algorithm does not succumb to out-of-order processing, we know that the completion metadata mapping can be constructed greedily without caching incorrect optimal derivation vectors.

The **top K queue** is useful in circumstances where we specifically are interested in only the top k parses of a given text, and we are interested in parses which do not necessarily cover the whole text. For example, we may give the parser an entire paragraph and be interested in the optimal parses for the individual sentences in the paragraph. If we are permitting there to be multiple completed sentential forms for a given text, then there are worst case $O(n^2)$ sentential form states to consider. We know, at the time of completion, however, what the utility of the best parse is for each state, and therefore we can tell which states definitely do not contain parse trees in the top k parses. Thus it is useful to maintain a priority queue consisting of the top k sentential form states, so that when we construct our forest only states that could potentially have the top k parses are counted in. Note though, that it's possible that one state contains all k best parses, thus maintaining a queue of the top k states is only know which states are *definitely* dead ends, it doesn't actually provide the top k parses or even tell them which state they are derived from. The top K parse queue can be maintaining a reverse priority queue whose first item indicates the worst state known to be in the top K so far. During each completion of a sentential form state S, if the top K queue is still smaller than size k we add S, otherwise we peek the top of the queue to reveal the $k^{th} Best$ state, and replace it with S if S is has a higher best parse value.

Under this policy we perform $O(log(n))$ additional operations and maintain $O(k)$ extra space. This method guarantees that we construct our SPPF out of only those states which could possibly contain the top k parses extractable from the input text.

4 Complexity

4.1 Time complexity

4.2 Space complexity

References

[1] J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 13, no. 2, p. 94–102, 1970.

[2] J. Goodman, "Semiring parsing," *Comput. Linguist.*, vol. 25, pp. 573–605, Dec. 1999.

[3] S. M. Shieber, Y. Schabes, and F. Pereira, "Principles and implementation of deductive parsing," *J. Log. Program.*, vol. 24, pp. 3–36, 1995.

[4] E. Scott, "Sppf-style parsing from earley recognisers," *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 2, pp. 53 – 67, 2008. Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007).

[5] P. Boullier and B. Sagot, "Are very large context-free grammars tractable?," *Proceedings of the 10th International Conference on Parsing Technologies - IWPT 07*, 2007.

[6] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to automata theory, languages, and computation*. Pearson Education Limited, 2014.

[7] G. F. Luger and W. A. Stubblefield, *AI algorithms, data structures, and idioms in Prolog, Lisp, and Java*. Pearson Education, 2009.

[8] M. Tomita, "Efficient parsing for natural language," 1986.

[9] B. v. d. Sanden, *Parse forest disambiguation*. PhD thesis, Technische Universiteit Endhoven, 2014.

[10] A. Lavie and M. Tomita, "Glr* – an efficient noise-skipping parsing algorithm for context free grammars," 1993.

A Attribute and utility example

Chapter 3. Building an ordered Shared Packed Parse Forest for noise-skipping Earley parses with tree-attribute based utility functions

Abstract

In the previous chapter I showed a handful of modifications necessary to retrofit the standard Earley algorithm plus Scott's [1] SPPF-building algorithm to perform parsing over noisy input streams. Even more importantly, the algorithm presented in the previous chapter is capable of maintaining a list of attributes $X(S)$ for each state S such that the attributes $X(S)$ are those of the best tree derivable from S, where best is defined by some utility function $f() : \mathbb{R}^N \rightarrow \mathbb{R}$. Chapter 1 provided conditions on the types of attributes and the types of utility functions for which the algorithm provided in the previous chapter is valid. Having an algorithm which performs noise-skipping and correctly maintains the optimal set of attributes among all parses derivable from each state is useful because it not only tells us the best possible derivation of a given sentential form but it also allows us to build the SPPF in an ordered manner, such that extracting the trees of the SPPF in order is possible polynomial in the number of desired trees and not in the number of total trees. This is very useful because, though SPPFs are polynomial in the size of the input sentence, they can contain exponentially many trees and extracting the trees in some rational order is a crucial prerequisite to performing any useful analysis of the results.

Before I can give an algorithm for extracting trees in order and discuss the applications for such an algorithm, I must describe the modifications necessary to build an ordered-SPPF out of the predecessor/reduction graphs created by the parsing algorithm described in the previous chapter. I will do so by first showing Scott's [1] original algorithm for building SPPFs from the predecessor/reduction graph of an Earley parse and then describing the modifications necessary to build ordered-SPPFs over predecessor/reduction graphs produced from noisy-inputs. In the next chapter, I will take up the subject of in-order tree extraction in polynomial time.

CONTENTS

Abstract 1
Contents 1
1 Introduction 1
1.1 The BuildTree function 1
2 Noise-skipping 2
2.1 Correctness of the labeling relations 2
3 Rank-ordering 4
3.1 Creating families 4
3.2 Adding children to an ordered-SPPF node 4
3.3 Updating families 4
References 6

1 Introduction

In the previous chapter I described the predecessor and reduction relations introduced by [1] to be computed at parse-time in order to enable full parse forest construction from an Earley parser. In this section I will describe the algorithm given in [1] for processing predecessor/reduction relations so as to build an SPPF corresponding to all and *only* the parses possible on the input. In the next section, I will give the modifications necessary to handle skipped noise (formally, to handle the differences in the predecessor/reduction labels described in Chapter 2 section 3.3) and in the following section I will give the modifications necessary to make the SPPF an ordered-SPPF, the usefulness of which will become clear in Chapter 4.

1.1 The BuildTree function

[1] describes a recursive function BuildTree which, given an SPPF node u and an Earley state p, populates u with children, each of which is an SPPF corresponding to a derivation anchored at p such that all possible subtrees anchored at p are stored in u. In this way, calling it on each sentential state returned by the parser will produce the overall SPPF for the input. The pseudocode below describes Scott's BuildTree function in a manner consistent with the notation used elsewhere in this thesis. For SPPF nodes, I use a notation (A, i, j) where A is the name (it can either correspond to a symbol in the vocabulary or a partially completed rule), i is the start index, and j is the end index. Every time a new SPPF node is created in the algorithm below, by convention, note that we first check to see if an identical node has been created thus far and instead use the original instead of producing a new copy. E.g. $v \leftarrow (a, i, i + 1)$ is shorthand for checking a global store to see if $(a, i, i + 1)$ exists and instantiating v to that node if it exists, otherwise instantiating it and adding it to the store.

Note that capital letters indicate nonterminals and indexed lowercases represent terminals, additionally greek letters represent sequences (non-empty) of symbols in the vocabulary. Family({w,v}) indicates the construction of a "family" which contains the set of SPPF nodes w and v.

Refer to [1] or [2] to further understand what precisely an SPPF looks like and how this algorithm produces them. Briefly, this algorithm creates terminal nodes for each explained input element, from there, every time it encounters a state which has @ anywhere but the first index, it

```
1:  procedure BUILDTREE(u, p)
2:      if p already processed then return
3:      u ← current SPPF node
4:      p ← current Earley state
5:      if p = (A, [a_i, @, β], [i, i + 1]) then
6:          v ← (a, i, i + 1)
7:          u.addChild(Family({v}))
8:      else if p = (A, [C, @, β], [i, j]) then
9:          v ← (C, i, j)
10:         for q ∈ p.reductions(i) do
11:             u.addChild(Family({v}))
12:             BuildTree(v, q)
13:     else if p = (A, [α, a_{j−1}, @, β], [i, j]) then
14:         v ← (a, j − 1, j)
15:         for p′ ∈ p.predecessors(j − 1) do
16:             w ← (A := α@a_{j−1}β, i, j − 1)
17:             BuildTree(w, p′)
18:         u.addChild(Family({w, v}))
19:     else if p = (A, [α, C, @, β], [i, j]) then
20:         for l ∈ p.reductions do
21:             for q ∈ p.reductions(l) do
22:                 v ← (C, l, j)
23:                 BuildTree(v, q)
24:                 for p′ ∈ p.predecessors(l) do
25:                     w ← (A := α@Cβ, i, l)
26:                     BuildTree(w, p′)
27:                 u.addChild(Family({w, v}))
```

creates either one or two children. If there is only one element before the dot then we create a single child SPPF node and build tree on it with every possible derivation of it (represented by iterating through all reductions from p at i). If there are two elements, we find every way to reduce the element before the dot, and then find ever predecessor explaining the state resulting from moving the @ backwards one position. Each family represents a way to split u's derivations into a pair of after-dot symbol and the result of moving the dot forwards by one index. Ambiguities occur because at some point in the forest a node has multiple families as its children - that is, the @ can be moved backwards in multiple ways. Thus the algorithm works somewhat in reverse relative to the algorithm used to construct the parse - we exhaustively explore all the possible ways to move the @ backwards per the reductions available and then all the possible ways to derive the resulting left children per the predecessors available, and right children (if the symbol is a nonterminal) via recursive calls to BuildTree. Note that the check on line 1 ensures we never process a state more than once.

2 Noise-skipping

In order to handle the modified predecessor/reduction relations associated with the noise-skipping algorithm, only 3^{rd} branch of the if statement needs to be modified substantially, an additional minor adjustment is needed in the 4^{th}. This is so because that is the only branch which relies on hard-coded labels on the reduction and predecessor relations. While the first and second branches are hard-coded as well, when the symbol being processed is the only thing before the @ its deterministic how to move the @ backwards, so the hard-coding is not modified by the introduction of the constraint that $j \neq j'$ during completion of states ending at j via states beginning at j'.

We end up modifying the pseudocode between lines 15 and 18 as well as modifying line 25 where we instantiate w in branch 4 - the updated pseudo-code is below:

```
1:  procedure NOISYBUILDTREE(u, p)
2:      if p already processed then return
3:      u ← current SPPF node
4:      p ← current Earley state
5:      if p = (A, [a_i, @, β], [i, i + 1]) then
6:          v ← (a, i, i + 1)
7:          u.addChild(Family({v}))
8:      else if p = (A, [C, @, β], [i, j]) then
9:          v ← (C, i, j)
10:         for q ∈ p.reductions(i) do
11:             u.addChild(Family({v}))
12:             NoisyBuildTree(v, q)
13:     else if p = (A, [α, a_{j−1}, @, β], [i, j]) then
14:         v ← (a, j − 1, j)
15:         for k ∈ p.predecessors do
16:             for p′ ∈ p.predecessors(k) do
17:                 w ← (A := α@a_{j−1}β, p′.i, p′.j)
18:                 NoisyBuildTree(w, p′)
19:                 u.addChild(Family({w, v}))
20:     else if p = (A, [α, C, @, β], [i, j]) then
21:         for l ∈ p.reductions do
22:             for q ∈ p.reductions(l) do
23:                 v ← (C, l, j)
24:                 NoisyBuildTree(v, q)
25:                 for p′ ∈ p.predecessors(l) do
26:                     w ← (A := α@Cβ, i, p′.j)
27:                     NoisyBuildTree(w, p′)
28:                 u.addChild(Family({w, v}))
```

Where the only change is that we look for all predecessor relations, some of which may be labeled with labels $k < j - 1$ as they may have resulted from moving the dot on a state q whose end index is less than $j - 1$, indicating noise was skipped. Additionally, we edit the instantiation of w to end at $p'.j$ instead of l as it may be the case that $l > p'.j$ and we want our SPPF nodes' indices to correspond to the indices of the Earley states that mirror them

2.1 Correctness of the labeling relations

In the previous chapter I noted the following generalization for how to produce the predecessor/reduction relations and how to label them during completion for the noise-skipping case:

> There are two cases: if nt had a terminal as its LHS then we still use j as the label for

> the predecessor edge from p to q, where j is the end index of q. If, however, nt had a nonterminal as its LHS then instead of using j, the end index of q, we use $j' \geq j$, the start index of nt as the label for all predecessor *and* reduction edges from p to q. It is a bit curious that the nonterminal case and terminal case behave differently, namely that the terminal case, just as in the standard Earley algorithm, still uses q's end index j to label predecessor edges while the nonterminal case pivots and uses j' to label both edge types.

To remind the reader of the meanings of those terms: q is a state whose dot is moved forwards by the state being completed, $q \triangleq (A, [\alpha, @, B, \beta], [i, j])$, nt is the state being completed, $nt \triangleq (B, [\gamma, @], [j', k])$ and $p \triangleq (A, [\alpha, B, @, \beta], [i, k])$ is the successor state of q obtained by moving the dot forward and updating the end index to be that of nt.

It should now be clear why, during completion via a terminal, we use j as the predecessor label. This case is handled in the modification to the third branch in NoisyBuildTree. q's end index is the last token explained, by convention, so we indicate the location of the predecessor of a p produced by a scan as the end index of the q which precedes it. We then undo this process by searching across all possible labels k (all of which, by the execution of the algorithm in the last chapter will be $\leq j - 1$) for the end indices of a q which could've produced p via scan.

It should also be clear why the labeling scheme from the previous chapter allows us to keep case 4 in the if-statement only slightly modified from BuildTree. In particular, we base the start index of the node v at line 23 on the label l (equiv. to j' in the excerpt above) of the reduction for the derivation it was part of. The node v corresponds to the state nt in the above excerpt, and thus it is parsimonious that the label j' be the label of the reduction as it tells us where the exact span of nt is. We don't need to label reductions this way in the terminal case above because terminals always span 1 token so it is immediately clear what the indices of the corresponding SPPF node should be.

Furthermore, it is important that we indexed the predecessor with label l (j' in the parlance of the excerpt above) instead of with $p'.j$ (which corresponds to j in the excerpt above). Otherwise, we would not have been able to find the p' (corresponding to q in the excerpt) associated with this label l unless we exhaustively traversed all the labels of the predecessor map like we did in the third branch. This would be costly because we would need to do so for each state in the reduction lists. It is important to note that p' may appear as a predecessor to p in multiple predecessor lists mapped under different labels l as the mapping is based on the possible q's (again, corresponding to nt in the excerpt above) which combined with it, a design which allows us to index directly by the index under which q lives in p's predecessor list.

Thus, reframing the foregoing discussion in the parlance of the previous chapter, due to noise skipping, the same q could be a successor to p via many different completed states nt, nt', $etc.$. From the predecessor/reduction maps we know a q and an nt can be combined to produce p if they were both mapped under the label j' (the start index of nt) during the completion phase of the parse. We map q by nt's start index because q may occur in many different completions with nt's with different start indices and thus may occur multiple times under different reduction labels in p. For example for the input $C\ C\ C\ D\ D\ D$, its possible $q = (sidenetwork, [C, @, sidenetwork, D], [1, 2])$ may be completed by an $nt = (sidenetwork, [C, D, @], [3, 6])$ or an $nt' = (sidenetwork, [C, sidenetwork, D, @], [2, 6])$ to produce $p = (sidenetwork, [C, sidenetwork, @, D], [1, 6])$. Thus when processing p we would find both nt under predecessor label 3 and nt' under predecessor label 2. We would also find q duplicated in p's reductions, once under label 3 corresponding to the completion with nt and once under label 2 corresponding to completion with nt'. This duplication of q, resulting from indexing the reduction labels by j' in the previous chapter allows us to directly find all q's which helped combine with nt to produce p without the need to traverse all the keys of the predecessor list.

Storing q multiple times under the start indices j' of the nt's they can complete with makes for simpler code - allowing our fourth branch in NoisyBuildTree to be essentially identical to the corresponding branch in BuildTree. This scheme yields a slight speed boost over its alternative in the case when the reduction label keys aren't sorted, where the alternative is storing each q only once, by its end index j, furthermore it incurs greater memory overhead by duplicating mappings for q in the reductions map. Storing q once, by its end index j would require us to search all label keys j in the reduction map and then search for q's only in the lists whose label is $j \leq j'$. It is always the case that q stored in the reductions of p under label $j \leq j'$ can combine with nt, but if our label keys are not sorted we would need to check all reduction labels and then search only the lists whose $j \leq j'$. This would only yield a speed-up in the unlikely circumstance where the skip width w is very large but there are only on order $k \ll w$ states per label in the predecessor/reduction maps - thus the primary benefit is to make the code look as similar to Scott's code as possible, achieving greater programming simplicity in the process.

I believe the memory-code readability trade-off is worth the memory cost because the memory cost is likely small in the average case. The alternative reduction labeling formalism would have, in a loose worst-case analysis overhead of $O(|S|^2)$ where $|S|$ is the number of states, because in theory each state $|S|$ could have on the order of one mapping per state stored in its prediction/reduction maps. In this case, however, each $|S|$ could have one mapping per state in its reductions followed by one mapping in its predecessors per state for each reduction because the q's are duplicated for every reduction, this results in overhead of $O(|S|^3)$. In practice, even the longest parses aren't very long making $|S|$ relatively small - thus memory is cheap but code-readability is expensive, yielding a

personal preference for the formalism presented here, as opposed to the alternative.

3 Rank-ordering

While the modifications needed for building SPPFs robust to noise were relatively minor - namely the changes to the labeling relations introduced in the previous chapter and the brief modification of lines 15-18 and line 25 in BuildTree, the modifications to produced an ordered-SPPF are somewhat more substantial - relying both on the introduction of metadata maintained in the previous chapter and modification to the simple "set" architecture of the Family data structure introduced in [1].

There are four primary modifications required in order to make the SPPF ordered. Before explaining those modifications I will briefly define ordered-SPPFs. In an ordered SPPF, each node u has associated with it the best Earley state associable with it S, where best is determined by applying the utility function to the attributes $X(S)$ - again, reminding the reader that $X(S)$ itself is the set of attributes associated with the best parse derivable from S. Furthermore we need to keep the families up to date with the best possible derivation corresponding to the SPPF nodes $\{w, v\}$ they consist of, finally we need to keep the families ordered in order of decreasing utility.

3.1 Creating families

When creating families we most automatically set the attributes of the best possible parse derivable from the state resulting from the combination of the elements in the state set (of which there are either 1 or 2). This is done via reference to the completion metadata described in the last chapter. To remind the reader, the completion metadata stores a mapping from tuples "(nt, q) to the optimal vector of attributes $X(p)$ which is completed by combining nt and q". Thus when we create the family F consisting of the set (nt, q) we can set the attributes and utility of F by evaluating the metadata map at (nt, q).

Pseudocode for the family constructors are given below:

```
1: procedure FAMILY(v)
2:     f.members = {v}
3:     f.states = {v.state}
4:     f.attributes = v.state.attributes
5:     f.utility = util(f.attributes)

1: procedure FAMILY(w, v)
2:     f.members = {w, v}
3:     f.states = {w.state, v.state}
4:     f.attributes = completions(f.states)
5:     f.utility = util(f.attributes)
```

3.2 Adding children to an ordered-SPPF node

In order to add children to an ordered-SPPF node we must have the underlying data structure for the child list be sorted somehow. A sensible implementation for the underlying data structure is a max-heap because, as we will see in the next section we will need to be able to update the utility value of families during the course of processing, and for heaps, ensuring proper heap ordering after modifying the priority of an element is as simple as deleting it and re-adding it with the new priority value. Thus the addChild() function is as simple as constructing the family in the manner given above and then adding the family to a max-heap of child families whose ordering predicate is the utility of each family.

3.3 Updating families

Finally, we must be sure to update families when better attributes become available for them. In the pseudocode for BuildTree we always simply added new families to the child set of each node, but now we must update the attribute values of families if the family already existed in the max-heap with a lower utility score than the current instantiation. Thus in order to properly maintain order we need to account for the possibility that a family's optimal attributes may be changed due to it being created again at some point. Thus a necessary update to our code is that instead of merely adding the new family to the child set of the current family, we need to check to see if a family with the same signature already exists, pull it out, merge it with the new one using the procedure below entitled UpdateFamily and put the result in the children max-heap.

```
1: procedure UPDATEFAMILY(u, v)
2:     f' ← Family(v)
3:     for f ∈ children do
4:         if f = f' then
5:             if f'.utility > f.utility then
6:                 u.children.remove(f)
7:                 u.children.add(f')

1: procedure UPDATEFAMILY(u, w, v)
2:     f' ← Family(w, v)
3:     for f ∈ children do
4:         if f = f' then
5:             if f'.utility > f.utility then
6:                 u.children.remove(f)
7:                 u.children.add(f')
```

Thus our final function BuildTree'(u,p), incorporating the maintenance of best states and updating families when encountering them again is given below:

Note that the above procedure only applies when the optimal attributes for a family are updated via reinstantiating the family. One may also conjecture, that it is possible for a family's best attributes to be updated by a change to the optimal state associated with one of its members. This could be possible if when an SPPF node w were created for the first time, the state producing it was not the best possible state to produce it AND that the suboptimal value of the SPPF node's attributes get cached as the best value when added as a family to some upstream node u. If we later discovered a better derivation of w after we have exited the scope governing u then we would be unable to update the best family values for u, creating inconsistent "best family" attributes - namely,

```
1:  procedure BUILDTREE′(u, p)
2:      if p already processed then return
3:      u ← current SPPF node
4:      p ← current Earley state
5:      u.state ← argmax_{x∈{p,u.state}}(util(x.attributes))
6:      if p = (A, [a_i, @, β], [i, i + 1]) then
7:          v ← (a, i, i + 1)
8:          if Family({v}) ∈ u.children then
9:              UpdateFamily(u, v)
10:         else
11:             u.addChild(Family({v}))
12:     else if p = (A, [C, @, β], [i, j]) then
13:         v ← (C, i, j)
14:         for q ∈ p.reductions(i) do
15:             BuildTree′(v, q)
16:             if Family({v}) ∈ u.children then
17:                 UpdateFamily(u, v)
18:             else
19:                 u.addChild(Family({v}))
20:     else if p = (A, [α, a_{j−1}, @, β], [i, j]) then
21:         v ← (a, j − 1, j)
22:         for k ∈ p.predecessors do
23:             for p′ ∈ p.predecessors(k) do
24:                 w ← (A := α@a_{j−1}β, p′.i, p′.j)
25:                 BuildTree′(w, p′)
26:                 if Family({w, v}) ∈ u.children then
27:                     UpdateFamily(u, w, v)
28:                 else
29:                     u.addChild(Family({w, v}))
30:     else if p = (A, [α, C, @, β], [i, j]) then
31:         for l ∈ p.reductions do
32:             for q ∈ p.reductions(l) do
33:                 v ← (C, l, j)
34:                 NoisyBuildTree(v, q)
35:                 for p′ ∈ p.predecessors(l) do
36:                     w ← (A := α@Cβ, i, p′.j)
37:                     BuildTree′(w, p′)
38:                 if Family({w, v}) ∈ u.children then
39:                     UpdateFamily(u, w, v)
40:                 else
41:                     u.addChild(Family({w, v}))
```

the attributes for any family containing w would not actually be accurate. Inconsistency can also arise if after u has generated w, another state which is a parent of w in the forest, u′, creates it and w still does not have its optimal state associated with it. It turns out that neither of these scenarios are possible, as long as there are no cycles in the SPPF (a condition which underpinned the validity of utility functions in the first place in chapter 1).

The processing problem above corresponds to the out-of-order processing that we saw in chapter 1. This could happen in several places. First, on line 25 of NoisyBuildTree when a child of u, w is generated via a predecessor link, we could process w via a suboptimal state p′ leading to w have a suboptimal value associated with it when its family (w,v) is added to u. Even worse, we could imagine a new parent node u′ creates a reference to a family containing w while w still has a suboptimal state, and then finds the optimal state for w. This would create multiple families containing w in the forest which have different optimal values for w's state.

In fact, w can initially have a suboptimal state associated with it, but note that EVERY possible state that generates w exists in the predecessor list p.predecessors(k) and therefore, will be processed before u leaves scope. This means that initialization calls at line 25 resulting in suboptimal states will be overwritten by the time the scope for u is gone - therefore calls to UpdateFamily(u,w,v) will have the opportunity to fix the out of order calls before u exits. Furthermore, all possible derivations of w will have been explored before w is created again (i.e. before another possible parent u′ tries to instantiate w). This is always true as long as there are no cycles (that is, w is not created again during a recursive call happening while the call at line 25 is still on the call stack) - because of the reason given before: that before u leaves scope every possible state generating w will have been explored because every such state exists in the predecessors list. Note that in the case where there are cycles, and w may be encountered again while the call on line 25 is still on the call stack, the grammar must contain unary cycles and therefore is not valid for use with utility functions (see chapter 1).

The same logic applies at lines 34 and lines 37. In theory, either v or w could be first produced via suboptimal states - but every possible state generating v and w will exist in the appropriate reductions and predecessor lists, enforcing the constraint that by the time either of the child nodes w, or v are processed again in the future (by different parents, u′, u″, etc.) all of their possible generating states will have been explored - ensuring that the optimal state for each is found during the loop on line 32. Furthermore, u will not have left scope by the time all possible families resulting from different states deriving w and v are processed and, using the update family function, we will ensure that u leaves scope with the correct family attributes associated with any of its child families (w,v).

References

[1] E. Scott, "Sppf-style parsing from earley recognisers," *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 2, pp. 53 – 67, 2008. Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007).

[2] B. v. d. Sanden, *Parse forest disambiguation*. PhD thesis, Technische Universiteit Endhoven, 2014.

Chapter 4: In-order k-best parse extraction from an ordered-SPPF

Abstract

In the previous chapter we took up the task of generating an ordered-SPPF which was amenable to fast, in-order extraction of the k best trees. The only ordering constraint required to accomplish this task was that for each node in the forest u with family set $\mathfrak{F}$, each family $f \in \mathfrak{F}$ has associated with it, a vector denoting the attributes of the best possible subtree derivable from f.

I will show a motivating example of why in-order, k-best extraction is useful even for grammars which are incredibly simple, and then I will provide 3 algorithms: $O(1)$ best-tree extraction, brute force k-best extraction, and $O(k^2 log(k))$ k-best extraction.

This work presents a major contribution to this field as it is the only work which addresses polynomial time extraction of trees statically from a forest which has been built using utility functions. This is in part because work in the field primarily focus on dealing with optimality on the front-end, that is designing the parsing algorithm or the forest building algorithm itself to extract best trees, or work using approximate best parsing. This model is far more robust as it presents a way to extract best trees from a static, non-modified forest object which is provably correct and complete from the theoretical standpoint of the modified Earley algorithm and SPPF algorithms presented in the previous chapters. That is, rather than allowing the design parameter "k" shape the results of the parse from the beginning of the parse on, the same SPPF object can be queried many times with different "k"s (or with no "k" at all, i.e. return all trees, in order) and still consistently convey correct results.

Statically querying a complete SPPF presents a benefit over existing methods as it is not typically burdensome to perform the parse or build the forest, per se - it is usually only burdensome when you want to sift through the results. Thus, this algorithm takes the approach that it is better to leave the parsing and forest-building steps complete and correct and prune our results ad-hoc, as they are requested.

Contents

Abstract 1
Contents 1
1 Related works 1
2 Definition and use case 1
3 Algorithms 2
3.1 O(1) best tree algorithm 2
3.2 Brute force in-order extraction 2
3.3 $k^2 log(k)$ k-best extraction 3
3.3.1 Formal definitions 3
3.3.2 Algorithm 6
4 Conclusion 8
References 8

1 Related works

2 Definition and use case

Consider the simple toy grammar below:

Grammar 1

$$E := E + E \quad (1)$$

$$E = 1 \quad (2)$$

This toy grammar is an excellent grammar in our setting because its use case is ubiquitous (see, every programming language specification ever) and it produces highly ambiguous parses. Ignoring noise-skipping for now, the number of parses for an expression in the language above is C_n where n is the number of 1s and C_n is the n^{th} Catalan number, which is exponential in n - by the time we've reached 25 1s we are already past 10^{13} possible parses. The blow up in the number of parses is even more astounding when one permits noise skipping, as we then get a number of possible parses on the order of $C_n + \binom{n}{1}C_{n-1} + \binom{n}{2}C_{n-2}$....

In either the noise-skipping case, or the non-noise skipping case it may be incredibly useful to extract trees in order of some utility function. For example, in the noise-skipping case, simply extracting trees in order of tokens explained would give us the first C_n complete parses before any of the lesser parses. Furthermore, one could implement an attribute function that keeps track of the depth of the LHS and the RHS of a tree (this obeys the optimality criterion of chapter 1) which would then allow disambiguation to make + either left-associative or right-associative - that is if the weight for the LHS depth were higher than that for the RHS depth, then returning the top tree would return the tree for which all additions associated to the left. If the SPPF were not ordered in any way and we were required to extract all possible trees before finding the right one, then we would run into exponential overhead. Finding the best tree, or any number of the top k best trees would be like finding a needle in a haystack, requiring exponential overhead.

Furthermore, it is not entirely transparent how an unordered SPPF tree would reconstruct the utility values for all of its trees, thus making it not even necessarily possible to use the naive method of extracting by brute force and then pruning the trees, unless you knew exactly what

tree you were looking for (or what property you wanted it to have - e.g. max tokens parsed etc.). Thus without an ordered-SPPF it would be essentially intractable to find the best tree even once we had all the trees extracted.

Thus it is immediately apparent that in-order tree extraction is absolutely crucial for making grammars like the one above workable within the framework of general Earley parsing and SPPF building and it is of interest to find a general method that allows us to have ALL the potentially trillions of trees available within the forest but access them in a reasonably quick fashion in-order of some predicate of interest to you (the utility function defined at parse time). This especially so as we may want to change our utility function later on and be able to reparse and get different results. This is not so easy if one were using hard-coded methods to filter the input as its read in.

Below you see graphs for the parsing and forest building times for noise-skipping parses of expression grammar inputs as a function of length. You can also see the results for the three tree extraction algorithms described below: the O(1) best-parse algorithm, the brute force k-best in-order algorithm, and the heavy duty $O(k^2 log(k))$ algorithm which performs reasonably well far beyond when the brute-force algorithm is no longer tractable, but generally comes with costly overhead.

3 Algorithms

As I mentioned in the abstract, the key property of our SPPF that enables in-order extraction is its "ordered" property, where an SPPF being ordered means that for each node in the forest u with family set $\mathfrak{F}$, each family $f \in \mathfrak{F}$ has associated with it, a vector denoting the attributes of the best possible subtree derivable from f. As discussed in the last chapter, it is possible to do so in such a way that all of the families have consistent and correct best-attributes, as long as there are no loops in the forest and out-of-order processing is otherwise avoided. Additionally, as part of our preprocessing, the families are also sorted via a priority queue at build-time.

This style of organization lends itself to a particularly simple and fast algorithm for extracting the (not necessarily unique) "best" tree from the forest. I call this algorithm the O(1) best tree algorithm.

3.1 O(1) best tree algorithm

This algorithm is O(1) in the sense that it treats the size of the tree as a constant, a sensible analytical tool which I use for the remaining two algorithms as well. This makes sense because it goes without saying that one can't build a tree in time faster than the size of the tree - e.g. if it has 10 nodes, you can't build it in time smaller than 10 operations. Thus, I treat the size of each tree (in number of nodes) as a constant (though it is O(N) if N is the size of the parsed input) - making the best tree algorithm's run time asymptotically equivalent to the time required to build the tree.

Due to the ordering property of the SPPF, the algorithm is incredibly simple:

```
1: procedure BESTTREE(root)
2:     t ← tree(root)
3:     if root.families.size > 0 then
4:         best = root.families.peek()
5:         for child ∈ best.members do
6:             t.addChild(BestTree(child))
```

Where $tree(root)$ is some constructor that produces an appropriate tree-node description from an SPPFNode, e.g. a node which contains the tuple (LHS, i, j), the vector of attributes associated with root, and then an empty list of children .

The above algorithm takes exactly $O(N)$ where N is the size of the input, but - as mentioned above - N is essentially a constant as all trees must be proportional to it in size, so I will consider it to be a constant, giving this algorithm a zero-overhead, $O(1)$ extraction for the best tree where time is spent solely on building the tree.

3.2 Brute force in-order extraction

This algorithm, as you may be able to guess from the name, consists of walking over the entire forest, building each and every tree with its attributes associated with it properly, and sorting the results. If d is the number of trees, then this takes $O(d \cdot log(d))$. In the case of the expression grammar, $d = O(Ne^N)$, and each tree is proportional to N in size, so we get that the run time is $O(N^3 e^N)$. Which is pretty abysmal - but it turns it out it has relatively low overhead and for inputs where there are not too many parses, say less than 10,000, it outcompetes the polynomial algorithm presented in the next section. By definition as well, when $k \approx d$ one might as well just use this algorithm over the $poly(k)$ algorithm presented in the next section.

The algorithm relies on maintaining through recursive calls the current optimal vector for the root tree if we were to only take the best path down through the remainder of the forest (that is, the first family in every priority queue we encounter), and when we branch off into an ambiguity via an alternative family, creating a new recursive call and calculating the attribute vector that results from choosing the alternative family over the first family available in the priority queue.

The DiffVal() method below calculates the result on the optimum attributes vector if family f were chosen over family $best$. The implementation is somewhat specific to the attributes if noise-skipping is on and the two families cover different spans of the input. In particular, attributes such as noise skipped and tokens skipped are updated in a non-standard way if the families cover different spans. For the sake of brevity, I just write the pseudocode for the

general form, ignoring the non-standard updates for any attributes which may need special update rules when the families have different spans.

The AddResults method performs the following for each SPPFNode *node* and each family f of *node*: takes in a list of lists, where each sublist is all the possible subtrees resulting from an SPPFNode member of f, and in sum the overall list contains one sublist per member of the family. We then generate one new parent tree per way to choose one child tree per sublist (that is, the cartesian product of all possible subtrees for each member) and calculates the attributes of the resulting parent tree.

This calculation is non-trivial. Essentially, the vector passed in is the best possible attributes resulting from using that family of *node* (given the context of where in the recursive call stack where are at so far - that is, relative to the optimalVector parameter).

From there, for each child subtree whose best possible attributes are worse than the best possible attributes for *node* attainable by selecting family f (denoted *vec* in the pseudocode), we sum up the vector difference between the best outcome from that subtree and the best outcome for *node*, *vec*.

In the end, if the total difference plus the original vector *vec* passed in results in a worse set of attributes (evaluated under the utility function) than*vec* - that is, if choosing different children resulted in a suboptimal attribute set, then we set this t to have that vector, at the end we add the resulting t to a results list, accumulating all possible ways to generate the trees corresponding to *node* along with their attributes.

```
1:  procedure BruteForceForest(node, optimalVector)
2:      results_list ← []
3:      if node.families.size == 0 then
4:          t ← tree(node)
5:          results_list.add(t) return results_list
6:      else
7:          best = node.families.peek()
8:          for f ∈ node.families do
9:              newOptVec
10:                 ← DiffVal(best, f, optimalVector)
11:             alts ← []
12:             for child ∈ f.members do
13:                 possibleSubtrees
14:                     ← BruteForceForest(child, newOptVec)
15:                 alts.add(possibleSubtrees)
16:             AddResults(node, alts, newOptVec, results_list)

1:  procedure DiffVal(best, f, optVec)
2:      ret = optVec ⊖ best.vector ⊕ f.vector

1:  procedure AddResults(node, alts, vec, results)
2:      possibleChildren ← cartesianProduct(alts)
3:      for children ∈ possibleChildren do
4:          t ← tree(node)
5:          totalDiff =← []
6:          for child ∈ children do
7:              if child.vector < vec then
8:                  totalDiff+ = child.vector ⊖ vec
9:              t.addChild(child)
10:         result = totalDiff ⊕ vec
11:         if result < vec then
12:             t.vector = result
13:         results.add(t)
```

3.3 $k^2log(k)$ k-best extraction

This algorithm is the only algorithm so far which can return more than one tree in time proportional only to the number of trees requested - NOT proportional to the total number of trees. Thus, for $d >> k$ this method must be used over the one given in the previous section because the brute-force algorithm gets no speed-ups from the user only requesting the first k trees.

This algorithm is somewhat unusual and has difficult formal properties to prove - it will also take many times reading through to fully understand. It relies primarily on the notion of extracting a tree from the forest by choosing the best available family at each step (at each SPPFNode), and then queueing up subtasks corresponding to different ways to explore the ambiguities at the frontier of the current best-families path. A substask loosely corresponds to asking the question: "what if instead of choosing the best family at the previous step, I chose some other family and from THEN on chose only the best families?".

Choosing an alternative family f' is equivalent to exploring an ambiguity in the forest. At the time of creating a subtask corresponding to some ambiguity, we calculate the utility that would result from taking alternative family f' in context, and from then on only choosing best families. We queue up these subtasks in a priority queue based on their utility, and then keep executing subtasks to completion until we have returned k trees.

Below is a pictorial representation of an ambiguity. At the time we have reached family f while executing the current task there is an alternative path forwards - f′. The subtask here corresponding to taking f′ would result in replacing the subtree t generated by taking the best path from f onwards with the subtree t′, generated by taking f′ here and then the best path from f′ onwards.

Similarly, I have displayed the subtrees corresponding to the two alternatives. Thinking about a subtask as being a way to generate a different subtree and graft it into the current working tree will help to motivate the terminology and algorithm below.

3.3.1 Formal definitions

I formalize some of the requisite notions below:

- A **tree** t is a tree in the usual sense, but note that due to the binarization of our SPPF forest, all of our trees follow the form given in figures 2 and 3, each child set of a node (S, i, j) consists of at most 2 children corresponding to a rule $S \rightarrow \alpha\beta\Gamma$ where the left child is always $(\alpha\beta@\Gamma, i, j')$ and the right

child is either (Γ, j', j) if Γ is nonterminal or simply Γ otherwise.

- A **tree expansion** *of an SPPF node* **n** is a tree resulting from creating a tree node corresponding to **n** and tree expanding the child families (if there are any) recursively according to *some* selection policy
- A **tree expansion** *of a family* **f** results in up to two subtrees generated by tree expanding the SPPF node members (of which there are either 0 or 1)
- A **thread of execution** *anchored at family* **f** is a process resulting in one or two trees which begins at some family **f** and performs its tree expansion under the policy which always selects the best family when multiple child families are present. I will denote the tree resulting from **T**, with $[\![\mathbf{T}_f]\!]$.
- A **subtask: S** is a 6-tuple (T,f,f',origin,parent,root) consisting of a current **thread of execution: T**, a **primary family: f**, an **alternate family: f'**, an **origin SPPFNode: origin**, a parent tree corresponding to **origin** called **parent**, and **the root's vector of attributes: root**.
  - **T** is the thread of execution that was being completed when the subtask was queued up, it is important because it determines what the context tree will be when **S** finally executes.
  - **parent** is the subtree into which the subtask's result will be grafted. For example, (E,0,5) is the parent in figures 2 and 3, under which the trees resulting from f' are placed.
  - **f** represents the family the family whose subtree under the expansion in **T** is being replaced by the results of **S**.
  - **f'** is an alternative family to **f**, much like in figure 1 - it represents an ambiguity and within the current thread of execution represents a choice resulting in an alternative tree.
  - **origin** represents the SPPFNode at which the ambiguity f/f' arose, f and f' are child families of **origin** - it is the node analogue of **parent**.

    Essentially, the subtask performs the tree expansion of f' and grafts the resulting subtrees into the children of **parent**. I denote the subtree/s resulting from evaluating S $[\![\mathbf{S}]\!]$ and indicate grafting $[\![\mathbf{S}]\!]$ in place of **parent**'s original children with the notation: $[\![\mathbf{T}_f]\!](parent/[\![\mathbf{S}]\!])$
  - **root** is the vector of attributes associated with $[\![\mathbf{T}_f]\!](parent/[\![\mathbf{S}]\!])$

  The thread of execution isn't actually necessary as an argument to construct subtasks, as it is implicit in the combination of parent, f, origin, and root. It also isn't necessary to pass in f, as it is only needed in order to calculate the differential value of choosing f' over f - something which can be done at subtask construction timer, so the pseudocode will not refer to the current thread of execution or f explicitly.

Thus the subtask definition above maps nicely onto the situation in figures 1-3: the thread of execution T is currently producing the tree corresponding to the root node, when it encounters f' it creates a subtask corresponding to spinning up a new thread of execution anchored at f' whose parent tree is $(E, 0, 5)$. Executing that subtask would result in figure 3 which is left-associative, per f', compared to figure 2 which is right-associative, per f. In general, the process of finding trees begins with a single thread **T** anchored at the best family of the root - all subsequent subtasks are anchored subsequent families and are either discovered along the primary master thread or as subtasks of an executing subtask, subtasks of a subtask of a subtask, etc.

Note however that the family f which is replaced by a subtask is not necessarily the family being expanded when the ambiguity is discovered - as is the case in figures 1-3. Instead, it is also possible to complete some current thread of execution anchored around some family f, and then back up and reconsider different ways its neighbor family g, g', g'', etc. could've been realized - taking the tree just produced during the thread of execution anchored on f and swapping the neighboring family g's subtree with the result of a subtask anchored around the ambiguities g', g'', etc.

There are 4 different types of subtask whose typology is determined by their relationship to the current thread of execution. This typology is crucial to understanding how an algorithm designed around discovering and executing new subtasks could possibly find all of the trees in a forest.

(1) Type 0 subtasks (or **twin subtasks**) arise between families which are in alternation of one another. An example is given in figure 1. If the thread of execution is on f then a twin subtask should be spun up for f', corresponding to choosing f' instead of f.

(2) Type 1 subtasks (or **descendant subtasks**) arise when an alternative family is at least two generations below the family on which the current thread $\mathbf{T}_f$ is anchored. More formally, if f is the family on which T is anchored, then g' is a suboptimal alternative to family g where g and g' are both child families of some SPPF node descendant of f which isn't a direct child of f. These ambiguities are discovered somewhere on the call stack of the recursive component of the tree expansion phase. An example is given in figure 4.

(3) Type 2 subtasks (or **child subtasks**) arise when an SPPF node which is a direct child of the family about which **T** is anchored has an alternative family relative to its best. See figure 5

(4) Type 3 subtasks (or **sibling subtasks**) occur only as a consequence of subtasks which are not the root thread of execution. They occur when the thread of execution is anchored on a family e' which is a child of SPPFNode n2 whose parent family f contains another SPPFNode n1 (what a mouthful! see figure

6 for concrete examples). The type 3 subtasks are spun up around the child families (or family) of n1 (e.g. d and d′). The terminology "sibling" is motivated by the notion that the family f is the parent family of e′,e,d, and d′ and e′ exists in a sort of sibling relationship w.r.t. d ad d′. Direct family relationships become complicated in the SPPF due to the fact that its a bipartite graph. If we imagine f as consisting of n1 and n2, as opposed to having n1 and n2 as children then it becomes more clear why e′ is a sibling to d and d′. In this case it is not clear what e's relationship should be to e′ though I think alternate or twin are serviceable terms.

(5) Type 4 subtasks (**aunt subtasks**) occur when the thread of execution is anchored on a family that is off the optimal path and its grand family (or great grand family, etc - this is a generalization of the notion of aunts, grand aunts, great-grand aunts, etc.) contains SPPFNode of which it is not a descendant (i.e. an aunt). An aunt subtask is spun up for every family of the aunt SPPFNode. For example, if a thread of execution is anchored on c, in figure 7, then f is its grand family and n2's two families e and e′ represent aunt tasks off of this thread of execution. The basic motivation for aunt tasks is that, if I have now decided to explore the alternative paths given by c, we need to now move back up the tree and explore different ways to have explored members of the tree *above* c. This is so because c is an alternate way to express the contents that would've been realized by g during the first thread of execution but it is necessary that once c is explored, we also explore the different ways to have generated g's cousins in case there are ambiguities below the aunts e/e′ which aren't capture on the current thread $\mathbf{T}_c$

The core logic behind queueing subtasks during processing is that it permits us to, rather than enumerate all possible trees by explicitly exploring every possible combination of ambiguities, as we are in the midst of exploring the tree resulting from some particular combination of ambiguities, we can also, for every time a best family is chosen we can also queue up - but not necessarily execute - the logic resulting from asking - given that I am choosing to pursue path x right now, what would happen if I were to merge those results with either going back and redoing a choice which has already been made (as is the case with types 2-4) or what if I were to make a different choice at the moment but keep all other results the same (types 0-1)?

One can think of the subtree logic as a form of lazy evaluation, constantly exploring new threads of execution which are on the frontier of the ambiguities explored thus far. The 5 types are necessary such that we are able to back-up and consider the implications of combining the current path we are in with any possible different choice of path anywhere else in the forest - with the 5 types we are able to make switches at the level of twins, children, siblings, and (via aunts) arbitrary ancestors, cousins, etc.

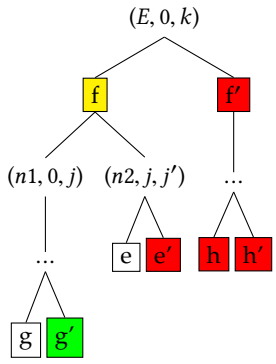


Figure 4: A type 1 subtask (or descendant subtask) is discovered for g′ deep in the recursion stack during the tree expansion of $(E, 0, k)$'s first family f - this is the root execution thread $\mathbf{T}_f$. The subtask corresponding to g′ has origin node n1 and is a descendant of f, the root of the thread of execution. Its parent tree will be the subtree $(n1, i, j)$ resulting from the thread's completion. It has primary family g and alternative family g′. Note that f′ is not a type 1 subtask because it is a sibling to f and that neither h nor h′ are type 1 because neither of them lie along the thread of execution (because f′ is not the best family of the root - f is). Finally, e′ is not a type 1 subtask because it is a child family of an SPPF node which is a direct child of f.

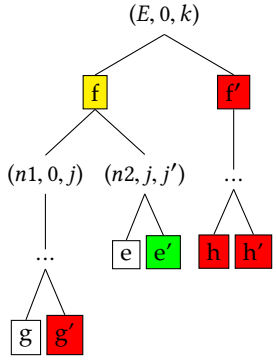


Figure 5: Only e′ is a type 2 subtask (child subtask) because it is an alternative family under a node which is a member of f (the anchor of the main thread $\mathbf{T}_f$)

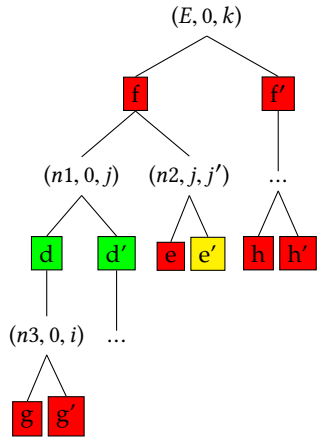


Figure 6: The thread of execution is anchored around e′ (indicated in yellow), in this case, d and d′ are sibling, or type 3, subtasks relative to the the thread of execution $T_{e'}$

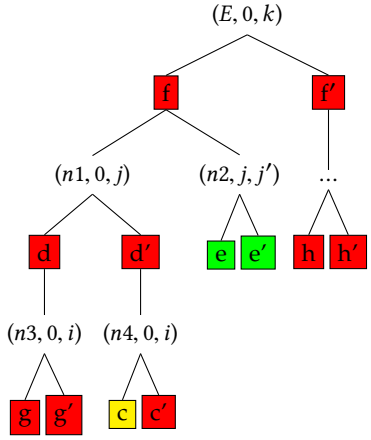


Figure 7: The thread of execution is anchored around c (indicated in yellow), e and e′ are its aunts - its necessary to be able to jump upwards in the tree - all the other subtasks shown so far simply permit us to jump downwards or sideways along the ambiguous branches of the SPPF

3.3.2 Algorithm

I now give the pseudocode for the subtask-based tree extraction algorithm. The algorithm begins with a root task based off of the best family of the SPPF root. It then queues up subtasks and puts them into a priority queue, popping them off in order of their utility. Due to the type 4 subtask, it is possible for duplications to occur, thus we must accumulate only the unique trees and terminate once we have obtained k unique trees.

```
1:  procedure ITERATIVELOOP(root, optVec)
2:      completed ← {}                                  ▷ Ordered set
3:      orig ← tree(root)
4:      best ← root.families[0]
5:      task ← (f′ : best, optVec : optVec, parent : orig,
6:          familyVec : optVec, origNode : root)
7:      q.add(task)
8:      while !q.empty() ∧ completed.size() ≤ k do
9:          next ← q.pop()
10:         t ← next.submit()
11:         completed.add(t)
12:     return completed

1:  procedure SUBMIT
2:      parent ← copy of parent tree
3:      root ← copy of root tree
4:      bestSubtrees ← {}
5:      for child ∈ f′ do
6:          childTree = tree(child)
7:          if child.isLeaf() then
8:              childTree.vector = child.vector
9:              parent.addChild(childTree)
10:         else
11:             optFam = child.families[0]
12:             childTree.vector = optFam.vector
13:             parent.addChild(childTree)
14:             recurse(optFam, optVec, childTree)
15:             bestSubtrees[optFam] ← childTree
16:     twinSubtasks()
17:     descendantSubtasks()
18:     childSubtasks(bestSubtrees)
19:     siblingSubtasks(parent)
20:     auntSubtasks(parent, root)
21:     return root
```

```
1: procedure RECURSE(fam, optVec, parent)
2:     for child ∈ fam do ▷ continue the tree expansion
3:         childTree = tree(child)
4:         if child.isLeaf() then
5:             childTree.vector = child.vector
6:             parent.addChild(childTree)
7:         else
8:             optFam = child.families[0]
9:             childTree.vector = optFam.vector
10:            parent.addChild(childTree)
11:            recurse(optFam, optVec, childTree)
12:            bestSubtrees[optFam] ← childTree
13:    for child ∈ fam do ▷ create type 1 subtasks
14:        childTree = tree(child)
15:        if ! child.isLeaf() then
16:            for altFam ∈ child.families[1 :] do
17:                primaryFam = child.families[0]
18:                diffVal =
19:                    DiffVal(primaryFam, altFam, optVec)
20:                origTree = bestSubtrees[primaryFam]
21:                taskParent = origTree.clone().rmvKids()
22:                task ← (f′ : altFam, optVec : diffVal,
23:                    parent : taskParent,
24:                    familyVec : altFam.vector,
25:                    origNode : child)
26:                task.root = this.root
27:                taskq.add(task)
```

```
1: procedure TWINSUBTASKS
2:     if origNode = null ∨ f′ = origNode.families[0] then
3:         return
4:     for altFam ∈ origNode.families[1 :] do ▷ type 0 subtasks
5:         primaryFam = origNode.families[0]
6:         diffVal =
7:             DiffVal(primaryFam, altFam, optVec)
8:         origTree = parent
9:         taskParent = origTree.clone().rmvKids()
10:        task ← (f′ : altFam, optVec : diffVal,
11:            parent : taskParent,
12:            familyVec : altFam.vector,
13:            origNode : child)
14:        task.root = this.root
15:        queue.add(task)
```

```
1: procedure DESCANDANTSUBTASKS
2:     queue.addAll(taskq) ▷ type 1 subtasks
```

```
1: procedure CHILDSUBTASKS(bestSubtrees)
2:     for child ∈ f′ do ▷ create type 2 subtasks
3:         if ! child.isLeaf() then
4:             for altFam ∈ child.families[1 :] do
5:                 primaryFam = child.families[0]
6:                 diffVal =
7:                     DiffVal(primaryFam, altFam, optVec)
8:                 origTree = bestSubtrees[primaryFam]
9:                 taskParent = origTree.clone().rmvKids()
10:                task ← (f′ : altFam, optVec : diffVal,
11:                    parent : taskParent,
12:                    familyVec : altFam.vector,
13:                    origNode : child)
14:                task.root = this.root
15:                queue.add(task)
```

```
1: procedure SIBLINGSUBTASKS(parent)
2:     if origNode.parentFamily = null then
3:         return
4:     for sib ∈ origNode.parentFamily do
5:         if sib = origNode then
6:             return
7:         sibTree ← getSiblingTree(sib, parent)
8:         for altSib ∈ sibling.families[:] do ▷ create type 3 subtasks
9:             primaryFam ← sib.families[0]
10:            diffVal ←
11:                DiffVal(primaryFam, altSib, optVec)
12:            taskParent ← sibTree.clone().rmvKids()
13:            sib.parentFamily.remove(sib)
14:            sib.parentFamily.remove(origNode)
15:            task ← (f′ : altSib, optVec : diffVal,
16:                parent : taskParent,
17:                familyVec : altSib.vector,
18:                origNode : sib)
19:            task.root ← this.root
20:            queue.add(task)
```

```
1:  procedure AuntSubtasks(parent, root)
2:      auntParent ← parent.parent          ▷ type 4 subtasks
3:      if auntParent = null then
4:          return
5:      backup ← auntParent.parent
6:      while backup ≠ null do
7:          for childTree ∈ backup.children do
8:              if childTree = auntParent then
9:                  continue
10:             aunt ← childTree.node
11:             for altAunt ∈ aunt.families[:] do
12:                 primaryFam ← aunt.families[0]
13:                 taskParent ← tree(aunt)
14:                 backup.replaceChild(childTree, taskParent)
15:                 diffVal ←
16:                     DiffVal(primaryFam, altAunt, optVec)
17:                 aunt.families ← []
18:                 task ← (f′ : altAunt, optVec : diffVal,
19:                     parent : taskParent,
20:                     familyVec : altAunt.vector,
21:                     origNode : aunt)
22:                 task.root ← this.root
23:                 queue.add(task)
24:             auntParent ← backup
25:             backup ← backup.parent
26:
27:
```

4 Conclusion

References

Chapter 5: Memory-efficient streamed Earley parsing

Abstract

This work revisits the noise-skipping Earley parsing presented in chapter two and adapts the core logic to be capable of streaming - that is, dealing with inputs of unbounded length, one token at a time and returning sentential forms as they appear. One important use case to motivate this approach is deployment in self-monitoring systems, where log entries may be written sporadically over the lifetime of the system and certain sequences of entries can be parsed into events for which some separate diagnostic system must be notified. Under a continuous stream model it is important to trim state to the minimum space required to parse all valid sequences.

In this chapter I will first give an overview of the core pieces of metadata from chapter two that need to be altered, discuss the procedural changes necessary to the core logic, and finally, discuss the algorithm necessary to keep track of which states are "active" and which states are ready to be purged from the parser's internal data structures.

In the end, the total memory consumption at any given time index t is proportional to the cube of the longest span active at any given time. That is, if $(D, [@, \delta, B, \mu], [i, j])$ is the active state with smallest i then memory consumption is proportional to $O((t-i)^3)$. This is asymptotically equivalent to the memory consumption for an Earley parser with fixed span of size $t - i$, and thus presents a lower bound for memory consumption. More importantly, the memory consumption, in practice, should not increase dramatically over time, as i, in most cases will likely be proportional to t in most uses.

Contents

Abstract 1
Contents 1
1 Standard Earley Parser memory consumption 1
1.1 The Chart 1
1.2 The *by_after_dot* map 2
1.3 The completions map 2
2 Core procedural changes 2
2.1 The "take" loop 3
2.2 Producing dormant states 3
2.3 Completions 3
2.4 Copying dormant states from the last "take" 3
3 The active-state algorithm 4
3.1 Correctness 4
3.2 Complexity 4
References 4

1 Standard Earley Parser memory consumption

A standard Earley implementation knows ahead of time how large its input is (for example, you give it a sentence at a time, rather than a word at a time) and therefore can initialize its data structures to some fixed size upon start-up. The primary data structure underpinning the logic is the Chart. Though standard deductive presentations of the Earley omit discussion of he organization of the underlying chart structure, procedural explanations such as that in [1] reveal how the chart operates under the hood to organize states and guide the processing order of the algorithm.

An important data structure alluded to in chapter two is the *by_after_dot* map. This map is used in the completion phase to find states whose dot can be moved. For example if $(D, [\delta, @], [j', k])$ has just been completed, we can directly access all the states which are anticipating a D and have end index less than j', e.g. $(A, [\alpha, @, D], [i, j])$. This data structure is not strictly necessary for a standard Earley parser. One can simply look at every state in chart j' and find all states whose symbol after the dot is D. When noise skipping is introduced, it becomes more worthwhile to introduce the *by_after_dot* map, because instead of doing the exhaustive search of just chart j' one would need to do a search for all charts $j \in [j' - skipWidth, j']$. We will see that in the streamed case the *by_after_dot* map is a *necessity* because we only store the charts for $[t - buffer\_size, t]$ but there could very well exist active states whose end indices are less than $t - buffer\_size$ and whose symbol after the dot is D. This is true even in the non-noise skipping case. Imagine that our buffer size is 100 and we have $(S, [@, A], [0, 0])$ and $(A, [a_1, a_2, ..., a_{101}, @], [0, 101])$. At this point, it is still valid to complete $(S, [@, A], [0, 0])$ but its chart (chart 0) will have *just been deleted* during the last take, and the current charts in memory will be charts 1 through 101.

Finally, the *completions map* was introduced entirely due to utility functions and thus has no analog in standard Earley parsing, though it is absolutely crucial both that it still serve its primary function in the streamed parser setting and that it be modified to be compact.

1.1 The Chart

The primary alteration to the chart is that instead of there being an array of charts of length $sentence.length + 1$, we create a circular buffer (or any bounded data structure) of size $buffer\_size$. Since we want to simultaneously use relative indices to access the circular buffer and absolute indices to name the states, we must also maintain a count of the circular buffer's offset to convert absolute indices $(t, t-1, t-2., .., 1)$ to relative indices.

During each loop (each time step) we also maintain an extra chart called the *dormant chart* that will be copied over to the "end" (relative) index at the next valid time step - how this is done will be covered in section 2.2, there is important nuance to this process that arises because we may enter a noisy span for an indeterminate amount of time such that the dormant scan chart produced at t is not copied over until $t + k$.

Finally, whereas a standard Earley implementation may associate one by_after_dot map with each chart (because note that the indices of the candidate states for completion matter), the streamed analogue must migrate that map to have global scope and be indexed first by the end (absolute) index of the state then by the symbol after the dot.

It is important to note that in theory, the buffer size could be 2 and the algorithm would still work. It is unclear whether there is any practical or theoretical performance gain from choosing any value larger than 2.

1.2 The by_after_dot map

As noted, the by_after_dot map has become decoupled from any particular chart and instead must now contain all active states. As such it is also a doubly-keyed hashmap, first by index, then by symbol after the dot. As such, all indices for which there is an active state, and all index/LHS pairs for which their is an active state must also be maintained.

The "contract" enforced in order to ensure correctness and compactness is that any state which could, at some point in the future be completed (that is - have its dot moved, either via a completion or a scan) must be in the map. I call these states "active" states. If this were not the case then some solutions would be lost and correctness would be broken. In order to ensure compactness, only those states which are "active" should be in the map at the end of each time step in the algorithm. The difficulty of ensuring correctness and compactness is that which states are active can only be calculated by implicitly maintaining a dependency graph of all the states. Any state which is actively scanning and hasn't exceeded the skip width is a leaf of the dependency graph - but there can be any arbitrary number of edges radiating out at any depth out from the leaves. That is - $s1 \rightarrow s2 \rightarrow ... \rightarrow sn$. The problem is made more difficult by the fact that the states can change at each time step (an issue I will address in section 2.2 and section 2.4) making it more practical to only implicitly maintain the graph by tracking dependency counts (similar to ref counts in garbage collection).

1.3 The completions map

The completions map is vital for calculating the best attribute values for SPPFNodes during the forest building process discussed in chapter three. It does, however, grow proportional to the number of states seen so far, which means that it must be flushed when completions are no longer needed. It is not entirely clear however what policy should be adopted to know when a piece of completion metadata is no longer needed. It may be the case that it is provably correct to remove a completion metadata entry as soon as one of the entries becomes inactive. If this were the case, it would be difficult to calculate this dynamically as we'd need to make the completions metadata map be configured to bey queried by single states at a time. A much simpler solution emerges however - recall that in chapter two it was noted that mappings from completion metadata keys (nt, q) to resulting states p are unique. Thus a piece of completion metadata can be uniquely scoped to the new state produced during the completion. This means that the global completion metadata map can be replaced by locally scoped maps which contain mappings $(nt, q) \rightarrow X(p)$, $(nt'q) \rightarrow X(p)$, $(nt', q) \rightarrow X(p)$, etc. Thus when a state p becomes dead and its memory is freed, the corresponding completions memory is freed as well. This modification makes it possible to maintain completion information right up until the moment it will never be used again and to implicitly clear it without having to directly find the entries to delete - as one would have to do if the map were global.

2 Core procedural changes

The procedure differs predominantly for three reasons:

(1) Noisy spans can't be precomputed and in fact, once we enter one we don't know when it is going to be over

(2) Typically when we complete a scan for a token at index i we place a completed state for that scan in chart entry $i + 1$ because the end index of that state is $i + 1$. Such is generally the case for all completions which ensue - their end index is $i + 1$ and they must be placed on the next chart. But in the streamed parsing case, there is not necessarily a next chart (though there is a chart full of dormant states to be carried over into the next time step) and thus completion must happen as a separate loop from that governing scanning and prediction, as it occurs (if it occurs at all - i.e. if a scan succeeded during this "take") on a different chart. Thus completions get migrated to a separate phase of the "take" to be after all predictions and scans are completed

(3) In the non-streamed parser having all of our future charts laid out ahead of time simplified scanning and prediction. Simply place a state corresponding to the prediction/scan on each chart entry $j, j + 1, ..., j + skipWidth$, with a distinct state created for each future chart. Furthermore, during completion we could look back into previous charts to see if there were any states whose j value is within $j' - skipWidth$. Since there are now a small number of past charts and no future charts stored - it is necessary to take all states awaiting a scan or a prediction at a later time step to be copied from time step to time step and killed when they

have skipped too far. Furthermore, it now becomes necessary book-keeping to maintain a newJ value for each state which has been copied, indicating its current absolute index (e.g. at time step t a state $(S, [a, @, a], [j, j+1])$ would have newJ t indicating it has been carried over $t - j - 1$ times). This problem, combined with reason number 2 above calls for us to introduce a dummy chart of dormant states to be carried over into the next time step

2.1 The "take" loop

The algorithm now proceeds a single time step at a time, where each time step corresponds to "taking" the next token in the stream. Thus a "take" consists of:

(1) Incrementing the index of the *lastToken* read and the index of the *end* of the buffer and potentially overwriting the buffer, additionally, clearing the dormant states chart to be an empty chart
(2) Modifying the noise maps if the current token being taken is a noise token
(3) Carrying over dormant states from the last take and filtering them if they have skipped too much noise
(4) A loop which performs all scans and predictions for the states in the current working chart (which have been copied over from the dormant states chart in the last part of the algorithm)
(5) A loop which performs completions resulting from the scan/prediction loop if there was a successful scan (i.e. if *end* passes *lastToken*)
(6) Finally, a clean-up portion which prunes states which are no longer active.

I will not elaborate on steps 1-2 as they are implementation-specific and entail relatively straightforward book-keeping.

I will explain how steps 4-5 create dormant states to be carried over into the next "take" in section 2.2.

In section 2.3 I will explain how step 5 correctly performs completions using the new *by_after_dot* map and using the newJ index calculated during step 4.

Finally, in section 2.4 I will explain how step 3 happens - a discussion I have deferred until the reader has a clearer concept of how newJ and the dormant states chart works.

Briefly, removing dead states in part 6 works by iterating over all states marked as dead in the previous iteration, identifying states which were dependent on them and decrementing their ref_count if any states ref_count goes to zero in this process it is marked as dead and then its dependents are processed, etc.

2.2 Producing dormant states

While reading tokens at time step t during step 4 of the "take" algorithm described above, states are read in order (order is given by the topological sort criteria from chapter one) from the current chart and dormant states are produced for two reasons:

(1) If the current token is a noise token, then for each state s read during step 4, s is mutated so that its newJ is incremented by one and then s is added to the current dormant states chart
(2) If for a given state s its newJ-j (that is, the number of times its been carried over) is less than the skipWidth, s is cloned and initialized with $newJ$ one higher than the newJ of its predecessor.

Following the second condition, a normal scan and prediction take place according to what kind of state s is. If case one occurs the loop continues, or if $newJ - j > skipWidth$ the state s is marked as "dead" and the loop continues.

Scanning and prediction are otherwise identical, except in scanning, if the state doesn't match the current token it is marked as "dead". In prediction, for each new state predicted based on s, a new state ns is created (as per usual) *and* s is marked as dependent on ns, incrementing the ref_count of s by one.

The above two cases cover the main locations that states truly die. States also implicitly die during completion and when dead states are removed in part 6 above.

2.3 Completions

Completions are performed following step 4 and are performed only if a successful scan occurred - in which case the *end* of the buffer has been incremented by one, while *lastToken* is still equal to t (this is why I said earlier that the minimum buffer size is 2 for this algorithm).

At each step, the states in the $charts[end]$ chart are popped off and either completed if they are a completion state (their dot is last), or added to the dormant states chart.

The completion method is otherwise modified only insofar as how it decides which states are now dead or deref'd (which I will describe in section 3). Additionally, it accesses the global by_after_dot map instead of what was original a by-chart map in the non-streaming algorithm - as described in section 1.2.

2.4 Copying dormant states from the last "take"

Copying dormant states into the current chart occurs in step 3 of the "take" process. Any state which was had their dot at the beginning of the rule - that is, states produced by a prediction in the last "take" - have their i and j (which are equal) reset to be *lastToken*, which I have also been referring to as t. Then any state s for which $newJ \neq lastToken$ must have been carried over 1 or more noisy spans and therefore must have its newJ set to lastToken. Doing so also requires rehashing it in any hashable data structures it occurs in, such as the by_after_dot map, and the hash set storing the inEdges for each state which depends on s.

3 The active-state algorithm

I have already given two examples of how states die: they try to scan a token but don't match they or they are being carried over from the dormant scans shart and have skipped more tokens than the skipwidth allows.

I have also explained that states marked as dead also deref their dependents during clean-up.

I will now note how dependencies are set up, and the places in the completer method in which refs are modified.

Dependencies are added in three places:

(1) predictor: when a state $s = (A, [\alpha, @, B, \beta], [i, j])$ predicts all the possible productions involving B as LHS, for each resulting new state $ns = (B, [@, \gamma])$, s is made dependent on ns and its ref count is incremented
(2) clone: when a state is cloned in case 2 of section 2.2 all states dependent on the original state are also made dependent on the clone and its ref count is incremented
(3) completer: when a state's dot is moved as a result of completion, all dependents of the original state add, as a dependent, the new state resulting from moving the dot forward - ref count here is not incremented, because typically the new state is replacing the old one

Dependencies are decremented in several places

(1) As mentioned earlier, when a state is marked as dead - in the remove dead states clean-up of step 6, ref counts get decremented for every dependent of the dead state
(2) If performing completion via a state $s = (D, [\delta, @], [j', k])$ - for every state $st = (A, [\alpha, @, D], [i, j])$ which will be completed by s, the ref count of st is decremented. This must happen because s was originally added as a dependency of st during prediction.
(3) If completion of st is occurring via s is a terminal symbol (resulting from a scan) then we decrement the ref count of all dependents of st. We do this because in this case item 3 in the list above doesn't apply, so no new dependency is being added in place of st
(4) If we are advancing the dot in st by 1 due to completion then for every dependent of st if ns (the result of moving the dot forward ins st by one) already exists in that dependent's dependencies, we decrement the ref count because item 3 applies from above, but we are NOT replacing the old state with a new one

For any state st having its dot moved by completion of s in case 4 above we mark st as dead - but we don't allow it to decrement the ref counts of its dependents during the "remove dead states" procedure because all the requisite decrements have already been accounted for in 4, noted above. We accomplish this by simply removing its references to its dependents so that the core logic of state removal doesn't get a chance to decrement them.

This makes sense, because in the case 3 for adding dependencies we have created a successor state ns which all dependents dep are now dependent on, in place of st. The ref count is not changed if ns is new - if ns isn't new, however, we decrement the ref count because that means whereas before $dep \rightarrow ns$ and $dep \rightarrow st$ we have now deleted st so one edge has disappeared. In the case where ns is new, we only had $dep \rightarrow st$ which has been swapped out for $dep \rightarrow ns$. It is important that we remove st's references to its dependents so as not to reap them during the "remove dead states" procedure - this is so because they're not *really* st's dependents anymore - they're now ns's, and in some sense st hasn't really died its just been replaced by ns

In case 2, however, we do allow st to keep its references to its dependents, making it possible to once again decrement the ref counts for entries in its dependent's once dead. This is so because in case 2, there is no successor ns to replace st (because the dot can't move anymore). The entire chain of logic extending from dep to st is now complete, and in a very real sense, st is truly dead. Thus st still needs access to its dependents because it is the sole owner of those dependencies (it hasn't shouldered off that responsibility to some successor state because the dot can't move anymore)

3.1 Correctness

3.2 Complexity

References

[1] G. F. Luger and W. A. Stubblefield, *AI algorithms, data structures, and idioms in Prolog, Lisp, and Java*. Pearson Education, 2009.

Appendix A: A fast filtering algorithm for massive context free grammars

ABSTRACT

Context free language are a formal class of language with broad applications in natural language processing and related fields. Though much of the CFL parsing literature has become more oriented towards statistical methods over the last decade, there still exist important use cases for robust non-statistical algorithms. In particular, there exist many use cases where we may be able to construct context free grammars but for which there doesn't exist enough hand-annotated data to train a robust statistical parsing algorithm. To solve edge case problems like this one, we must turn to classical non-statistical methods in CFL parsing. In this paper, we focus on the problem of tractability for large grammars. Our work is preceded primarily by that of Boullier and Sagot [1] who created practical parsing algorithms for large grammars by developing methods for ad-hoc filtering. That is, for a given sentence, how do we run our parsing algorithm on the smallest relevant subset of the grammar possible. To this end they developed a suite of fast filtering methods for context free grammars. We offer a variant of their basic filtering algorithm that accomplishes the exact same filtering (in terms of, given a grammar and input, what subset of the grammar can we restrict ourselves to) but with a 10-20 fold speedup. Furthermore, we test benchmark our algorithm against there's on grammars close to 10 times larger than those they originally studied. Our work is a major contribution because at the scale of grammar size under consideration in this paper, even Boullier and Sagot's suite of algorithms, which are the broadest, most recent contributions to the field, demonstrate unacceptable run times.

1 DEFINITIONS

Context-free languages are a formal class of language originally described by Noam Chomsky and later studied in great depth by computer scientists due to their interesting computational properties. For review see, [3].

Context-free grammars are a particular way of representing context-free languages expressed in the form of a rewrite system.

A context free grammar is typically defined as being a 4-tuple (V, T, P, S), where V is a Vocabulary of string symbols appearing in the grammar, T is a list of terminal strings which can appear in elements of the language, P is a list of productions of the form $A \rightarrow \alpha\beta\gamma\ldots$ where the left-hand side (LHS) is a single nonterminal (formally, an element in V/T) and the right-hand side (RHS) is a sequence (possible empty) of elements in V. Finally, S is a single element in V sometimes called the sentential form, or the start symbol. Any string accepted by the language must be generable by a series of productions which starts at the sentential form, hence the name moniker "start symbol".

It is a well-known property of CFGs that one can take any CFG describing a language L containing rules with empty RHS's and rewrite it into a CFG containing no such productions. The resulting language is identical to L except it does not contain the empty string, more formally $L' = L - \{\epsilon\}$.

We will concern ourselves with grammars that have been rewritten so that they have no empty productions, therefore any rule with n elements on its RHS must result in a terminal string at least n tokens long. We will use this property to filtering rules which are too long to be applicable to a given input.

2 INTRODUCTION

Historically, context-free grammars have been used as ways to describe human language. Due to the pressing need for natural language processing tools and the guaranteed polynomial parseability of all CFGs much ink has been spilled on parsing algorithms for CFGs and how to optimize them. The earliest such algorithm was the Earley parser [2]. Though later algorithms such as the GLR parser were discovered to be much faster by exploiting precomputed LR tables [5], much of the literature on fast parsing via judicious use of filtering is concerned with variants of the Earley algorithm because its data structures are especially amenable to removing rules from consideration on the fly (as opposed to the GLR algorithm which would require recomputing its LR table every time the rule set is updated).

In order to produce robust and effective grammars, in the 2000s many natural language systems dealing with CFGs concerned themselves with automatically generating rules at massive scales, rather than relying on what little robustness a handmade grammar could offer.

Massive context-free grammars are untractable because 1.) the run time of the Earley algorithm is proportional to the grammar size and 2.) dynamic programming algorithms such as the Earley algorithm end up blowing up their internal representations when attempting to maintain hypotheses about millions of rules at a time.

In general, recent approaches to improve robustness, speed, and scale in natural language parsing have been more oriented towards statistical methods, especially statistical dependency parsers. However, there is an area of research which approached the problem by devising clever ways to dynamically filter the applicable rules for a given input, making it so that for any given input, only a small fragment of the grammar is applied to it. This approach rules out only those rules that, by one way or another, we are certain could not apply. This methodology works because the Earley parser and its variants are relatively simple algorithms and have quick run times when run on grammars that don't require a lot of backtracking (that is, grammars where the number of inapplicable rules is not much greater than the number of applicable rules). The question then becomes, how do we, given a grammar G and a series of inputs s_i, dynamically alter the grammar G to get subgrammars G_i that parse s_i quickly and are easily restored to the original G after each input is processed.

In 2007, Boullier and Sagot [1] offered a suite of algorithms to perform such filtering. The article is an excellent resource, and to our knowledge is the largest published collection of filtering algorithms for CFGs. We will concern itself with one of the forms of filtering suggested in [1] called basic filering, or b-filtering.

We concern ourselves with b-filtering because, as opposed to the other forms of filtering discussed, it doesn't rely on adjacency information based on left/right-corner properties of the grammar and so it generalizes to perform correct filtering for noise-skipping parsers.

3 B-FILTERING

b-filtering, when applied to a grammar G and input s, works by removing all rules that contain terminals in their RHS that aren't featured in the input s. This method works because we know that any rule P with terminal α can only ever successfully be applied to a string containing α thus, when parsing s it is safe to remove P if $\alpha \notin s$.

Consider the grammar G below applied to the input $a\ b$:

$$S \rightarrow A\ B$$
$$S \rightarrow C\ B$$
$$A \rightarrow a$$
$$B \rightarrow b$$
$$C \rightarrow c$$

The b-filtering strategy would eliminate the final rule, leaving us with just rules 1-4.

The downside to Boullier and Sagot's algorithm is that it requires that we check every rule to see whether it contains any terminals not contained in our input. In particular, it runs in time runs in time $O(|G|\times|s|)$. For very large grammars, this can be an incredibly time consuming process. Thus, an ideal solution would be one which only touches those productions which we would like to include in our subgrammar. We call this design principle counting-productions-in as opposed to ruling-productions-out.

b-filtering is a particular algorithm that accomplishes what we will refer to as content filtering. The purpose of this paper is not to modify the logic of content filtering, but instead to provide a faster algorithm to perform it.

Boullier and Sagot's paper details two other filtering methods: a-filtering and d-filtering. a-filtering is a technique which involves precomputation of right-corners of all constituents and rules out nonterminal rules on the basis of whether rules beginning with some non-terminal have a valid right corner. d-filtering is a more advanced method involving dynamically constructed finite state automata.

All of Boullier and Sagot's tests involve b-filtering as a preprocessing step followed by some policy of a and d-filtering. The present work concerns itself with a method that achieves the exact outcome, in terms of which rules are filtered out and which aren't, of b-filtering, but in a much smaller number of steps. This method is called b-tree-filtering and will be described below.

In lieu of access to the grammars and specific example sentences they use in their work, we will forego direct comparison of b-tree-filtering with their benchmarks and instead compare directly to our own implementation of b-filtering. This comparison is valid because all of [1]'s benchmarks use b-filtering as a preprocessing step and thus their total run time is lower bounded by the b-filtering time. Therefore, since our b-tree-filters achieve the exact same resulting rule set and run much faster than simple b-filters, they could be used as valid preprocessing steps to more advanced filters such as the a and d-filters offered by [1].

Additionally, [1] measures the total precision of the resulting filtered grammar under a number of different strategies. That is, the number of rules left over afte filtering which are actually applicable to parsing the input. As mentioned, b-tree-filtering and b-filtering remove exactly the same rules, so doing a side by side comparison would not be meaningful. But just for the sake of parsimony we will include a calculation of the precision after filtering.

I will now briefly introduce the grammar studied in this experiment and discuss some of its properties, then present an algorithm which performs exceptionally well on large inputs.

4 OUR GRAMMAR

Our test grammar was produced by converting a Systemic Functional Grammar of English based on Halliday's grammar from the Penman project [4]

The resulting grammar has around 13 million rules. The total grammar size, measured by the sums of the right hand sides is 115,738,333. Compare this to the two grammars under study by [1] which had $\approx 500,000$ rules each and total size 1 million and 12 million, respectively. The structure of the Halliday grammar is relatively flat and much of the rules arise from different ways of permuting the otherwise equivalent RHSs. In particular, there are 8,975,867 unique RHSs and 35,228 unique sets of RHS elements.

A consequence of this structure is that it is particularly well suited to both length-based filtering when input sentences are short (a basis for filtering not mentioned in [1] because their system permitted ϵ-productions while we have decided not) and b-filtering. Note that length-based filtering could not be performed on a grammar containing empty productions.

The grammar is incomplete and can't handle all possible English constructions. For that reason we will be analyzing performance on the following sentences only:

- the distributor registered this domain
- this domain was registered by the distributor
- this domain in the account was registered by the distributor
- this domain in the account was registered by the distributor in the domain
- the use of the nicknames in the decryption key is evidence that the account was the distributor of these samples
- the use of the nicknames of this domain in the decryption key is evidence that the account was the distributor of these samples
- the use of the nicknames of this domain in the decryption key in the record is evidence that the account was the distributor of these samples
- the use of the nicknames of this domain in the decryption key in the record of the activity is evidence that the account was the distributor of the software
- the use of the nicknames of this domain in the decryption key in the record of the activity is evidence that the account

was the distributor of the software and that it was registered by the communication of the distributors

- the use of the nicknames of this domain in the decryption key in the record of the activity is evidence that the account was the distributor of the software and that it was registered by the communication of the distributors in these discussions

In order to get a sense of how many rules of this grammar are truly applicable for each input, we filter first by length, then by content, and finally, by removing nonterminals which were made unproductive by the previous two steps of filtering. Below is a graph relating length of the input to the number of rules left after all stages of filtering.

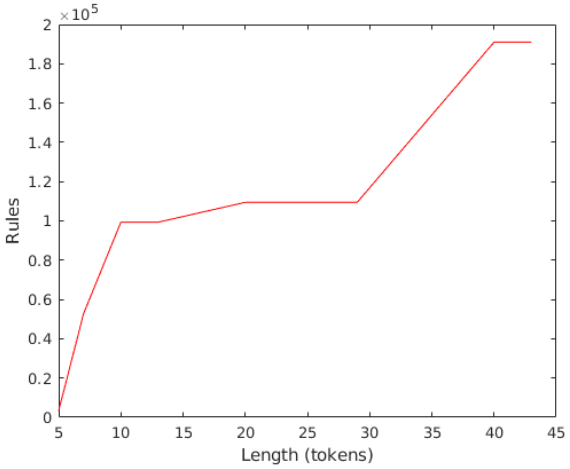


Figure 1: Number of rules resulting after content-filtering. Content-based filtering as exemplified in Boullier and Sagot's b-filtering and our b-tree filtering accomplishes close to 100 fold reduction of grammar size on sentences 45 words long

As you can see, our grammars respond incredibly well to content filtering.

We will now introduce a new algorithm that performs content-filtering which we call b-tree-filtering, and show how it compares to b-filtering, and a combination of length-filtering and b-filtering.

5 B-TREE-FILTERING

b-tree-filtering is motivated by the desire to accomplish the task of content filtering by only ruling-productions-in as opposed to b-filtering's method of ruling-productions-out. The key difference here, is that we want the number of operations performed to be proportional to the number of rules applicable *not* to the size of the entire grammar.

In order to do this we will construct a binary tree each node of which contains a set of rules indexable by the length of the RHS.

Each level of the tree will correspond to a terminal or preterminal in T. Thus we assume that there is an indexing a_i over the terminals/preterminals $a_0, \ldots, a_{n-1} \in T$.

The contents of the tree are defined inductively: the root, at level 0 contains all the rules. Each node at level i, if it has more than one rule, has its rules partitioned into two sets: those which contain a_i

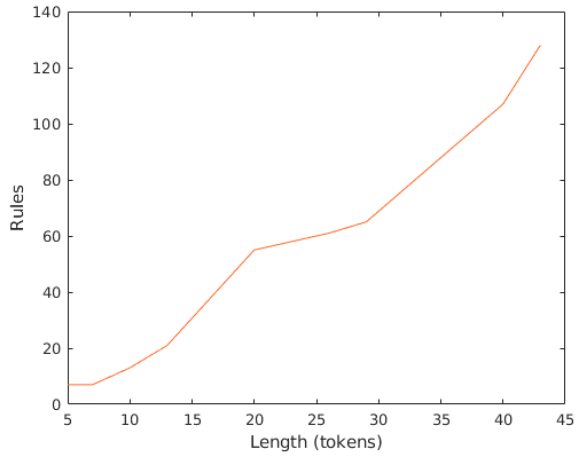


Figure 2: Number of rules used in successful parse (gold-standard). Though our content-filtering methods reduce the grammar to around 100,000 candidate rules, the actual parses for our sentences use on the order of 100 rules for long sentences, giving us relatively small precisions.

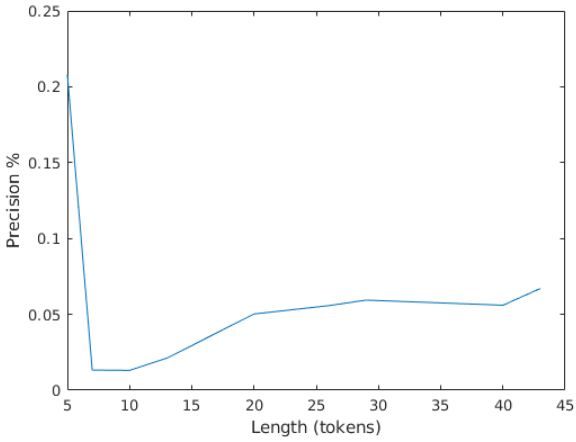


Figure 3: Precision in terms of % as measured by number of rules in the gold-standard divided by number of rules in the filtered set.

and those which do not contain a_i. If a node has only one rule, then it is set as a leaf node which stores just that rule. Structurally, what this means is, for all nodes at level i that have more than one rule (note "1 rule" is merely a stopping criterion we have pre-defined, we could've cut off branching at any cut-off number k) the left child (potentially null) consists of all those rules which do not have a_i and the right child consists of all those rules which do have a_i. Again, all rules are organized into sets indexed by length of RHS.

There are three structural properties we'd like to point out:

- Each level consists of some k nodes which are mutually exclusive and exhaustive of the set of all rules in the grammar.
- There are at most $|P|$ leaf nodes in the tree.
- If $|P| \gg |T|$ then there are $O(|P|)$ total nodes in the tree

The algorithm first calculates the maximum (or deepest) j of the a_i to rule out, then traverses the tree until it either reaches a leaf or passes level j. Once it reaches one of the above two terminating conditions, it extracts the rules in the right child (because these are the ones which do not contain any of the missing terminals/preterminals) on the present node, adds them to the working set and terminates that branch of the search, returning to the caller. In the case where we've reached a leaf node, we just check directly if its singleton rule contains any missing symbols in it.

The pseudocode below demonstrates the recursive search with $\cup$ denoting set union. In the case of length filtering included, then this is a set union over the sets whose RHSs are less than or equal to the sentence length, otherwise it's just a set union of all rules at that node.

Algorithm 1 b-tree-filtering

```
1:  max ← largest index missing from the input
2:  set ← ∅
3:  excl ← Symbols not in the input (missing symbols)
4:  root.GetRules(max, set, depth, excl)
5:  procedure GetRules(max, set, depth, excl)
6:      node ← this
7:      if node.depth ≥ depth then
8:          set ← set.rules.
9:      if node.rules.length = 1 ∧ depth ∉ ex then
10:         set ← set ∪ node.rules.
11:     if depth ∉ excl then
12:         set ← node.left.GetRules(max, set, depth + 1, excl).
13:         set ← node.right.GetRules(max, set, depth + 1, excl).
14:     if depth ∈ excl then
15:         set ← node.left.GetRules(max, set, depth + 1, excl).
16:     return set
```

The implementation we use can either simultaneously perform length-filtering on the fly or ignore length. This is true for both tree-based filtering and the standard b-filtering we benchmark against. This is because we store the rules in a Hashtable partitioned by length of the RHS and am able to choose to either only index rules of appropriate length or index rules of any length.

The largest rule is of length 12, so this technical point is irrelevant for sentences longer than 12 words, so we will not dwell on it too much. Furthermore, b-tree filter time isn't substantially impacted by whether length-filtering is used.

5.1 Results

Figure 4 contains timed benchmarks comparing filtering time for b-tree filtering coupled with length filtering, b-tree filtering without length filtering, b-filtering without length filtering, and b-filtering coupled with length filtering. Note that b-tree-filtering and b-filtering both have relatively stable run times as a function of sentence length for the sentences tested here. At short sentence lengths b-filtering is competitive with b-tree-filtering only when combined with length-filtering. In figure 5 we see that in all length domains studied here b-tree filtering is at least 7 times faster than b-filtering thus showing an extreme improvement over the simpler approach to content-based filtering presented in [1]. A very important point is that our filtering speed remains under a second, even for incredibly long sentences, while Boullier and Sagot basic filtering algorithm exhibits unacceptably large latencies of as long as 8 seconds.

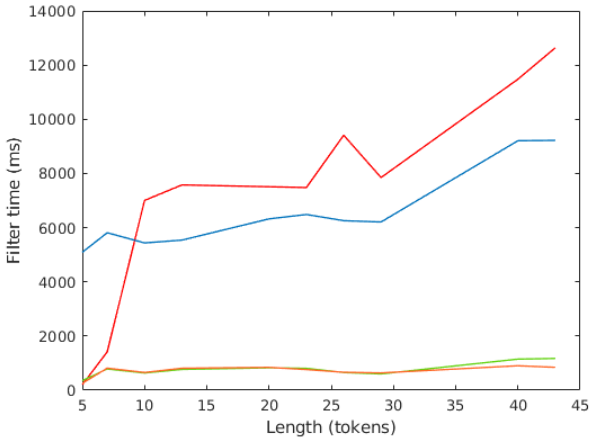


Figure 4: Filtering time in (ms) for sentences of varying lengths using the four methods detailed here. Orange: b-tree filtering with length filtering, Green: b-tree filtering without length filtering, Red: b-filtering with length filtering, Blue: b-filtering without length filterings. Note the approximate 8-fold speedup of b-tree filters in the large sentence limit.

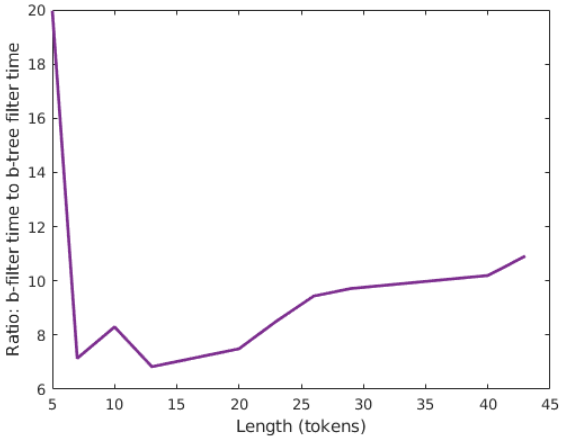


Figure 5: At least 7-fold speedup when using b-tree filtering with no length filter compared with [1]'s b-filtering. Our method does increasingly well for longer sentences.

The asymptotic relationship between run time of the b-tree filter and the number of rules arrived at in the final set appears roughly linear for our example grammar, though we lack a sufficiently large number of samples from the language to truly test that hypothesis.

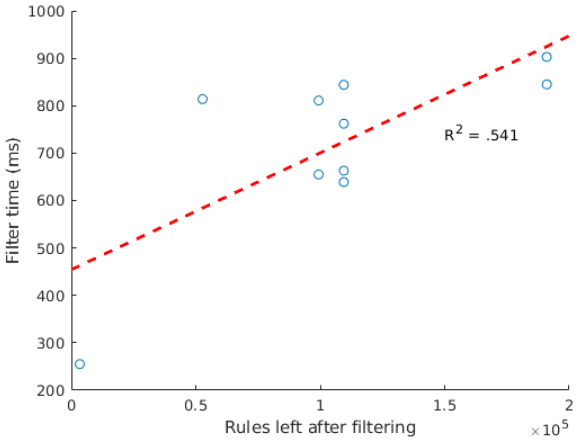


Figure 6: Run time of b-tree filtering grows approximately linearly in the number of rules in the resulting filter set, though the R^2 value is low and more robust sampling of the CFL would be need to show a strong relationship.

6 CONCLUSION

This work has demonstrated an extension to the most recent literature of filtering techniques for massive context free grammars. We have provided a benchmark comparison of a new algorithm, b-tree filtering, which is equivalent to the preprocessing filter used in [1]. Figure 7 show our algorithm runs anywhere between 7 and 20 times faster than Boullier and Sagot's basic filter on a grammar roughly 10 times the size of those investigated in their research. We have shown that our algorithm provides reasonable filtering times even for incredibly long input sentences.

REFERENCES

[1] Pierre Boullier and BenoÃőt Sagot. 2010. Are Very Large Context-Free Grammars Tractable? *Text, Speech and Language Technology Trends in Parsing Technology (2010)*, 201âĂŞ222. DOI:http://dx.doi.org/10.1007/978-90-481-9352-3_12

[2] Jay Earley. 1970. An efficient context-free parsing algorithm. Communications of the ACM 13, 2 (January 1970), 94âĂŞ102.

[3] John E. Hopcroft, Rajeev Motwani, and Jeffrey David Ullman. 2007. *Introduction to automata theory, languages, and computation*, Boston: Pearson Addison-Wesley. DOI:http://dx.doi.org/10.1145/362007.362035

[4] Eduard H. Hovy. 1993. Natural Language Processing by the Penman Project at USC/ISI. (January 1993). DOI:http://dx.doi.org/10.21236/ada269536

[5] Masaru Tomita. 1986. Efficient Parsing for Natural Language. (1986), 1âĂŞ7. DOI:http://dx.doi.org/10.1007/978-1-4757-1885-0_1