Parsing Linear Context-Free Rewriting Systems with Fast Matrix Multiplication

Shay B. Cohen[*]
University of Edinburgh

Daniel Gildea[**]
University of Rochester

We describe a recognition algorithm for a subset of binary linear context-free rewriting systems (LCFRS) with running time $O(n^{\omega d})$ where $M(m) = O(m^\omega)$ is the running time for $m \times m$ matrix multiplication and d is the "contact rank" of the LCFRS—the maximal number of combination and non-combination points that appear in the grammar rules. We also show that this algorithm can be used as a subroutine to obtain a recognition algorithm for general binary LCFRS with running time $O(n^{\omega d+1})$. The currently best known ω is smaller than 2.38. Our result provides another proof for the best known result for parsing mildly context-sensitive formalisms such as combinatory categorial grammars, head grammars, linear indexed grammars, and tree-adjoining grammars, which can be parsed in time $O(n^{4.76})$. It also shows that inversion transduction grammars can be parsed in time $O(n^{5.76})$. In addition, binary LCFRS subsumes many other formalisms and types of grammars, for some of which we also improve the asymptotic complexity of parsing.

1. Introduction

The problem of **grammar recognition** is a decision problem of determining whether a string belongs to a language induced by a grammar. For context-free grammars (CFGs), recognition can be done using parsing algorithms such as the CKY algorithm (Kasami 1965; Younger 1967; Cocke and Schwartz 1970) or the Earley algorithm (Earley 1970). The asymptotic complexity of these chart-parsing algorithms is cubic in the length of the sentence.

In a major breakthrough, Valiant (1975) showed that context-free grammar recognition is no more complex than Boolean matrix multiplication for a matrix of size $m \times m$ where m is linear in the length of the sentence, n. With current state-of-the-art results in matrix multiplication, this means that CFG recognition can be done with an asymptotic complexity of $O(n^{2.38})$.

---

[*] School of Informatics, University of Edinburgh, Edinburgh, EH8 9AB, United Kingdom.
E-mail: scohen@inf.ed.ac.uk.
[**] Department of Computer Science, University of Rochester, Rochester, NY 14627, United States.
E-mail: gildea@cs.rochester.edu.

In this article, we show that the problem of linear context-free rewriting system (LCFRS) recognition can also be reduced to Boolean matrix multiplication. Current chart-parsing algorithms for binary LCFRS have an asymptotic complexity of $O(n^{3f})$, where f is the maximal fan-out of the grammar.[1] Our algorithm takes time $O(n^{\omega d})$, for a constant d which is a function of the grammar (and not the input string), and where the complexity of $n \times n$ matrix multiplication is $M(n) = O(n^{\omega})$. The parameter d can be as small as f, meaning that we reduce parsing complexity from $O(n^{3f})$ to $O(n^{\omega f})$, and that, in general, the savings in the exponent is larger for more complex grammars.

LCFRS is a broad family of grammars. As such, we are able to support the findings of Rajasekaran and Yooseph (1998), who showed that tree-adjoining grammar (TAG) recognition can be done in time $O(M(n^2)) = O(n^{4.76})$ (TAG can be reduced to LCFRS with $d = 2$). As a result, combinatory categorial grammars, head grammars, and linear indexed grammars can be recognized in time $O(M(n^2))$. In addition, we show that inversion transduction grammars (ITGs; Wu 1997) can be parsed in time $O(nM(n^2)) = O(n^{5.76})$, improving the best asymptotic complexity previously known for ITGs.

1.1 Matrix Multiplication State of the Art

Our algorithm reduces the problem of LCFRS parsing to Boolean matrix multiplication. Let $M(n)$ be the complexity of multiplying two such $n \times n$ matrices. These matrices can be naïvely multiplied in $O(n^3)$ time by computing for each output cell the dot product between the corresponding row and column in the input matrices (each such product is an $O(n)$ operation). Strassen (1969) discovered a way to do the same multiplication in $O(n^{2.8704})$ time—his algorithm is a divide and conquer algorithm that eventually uses only seven operations (instead of eight) to multiply 2×2 matrices.

With this discovery, there have been many attempts to further reduce the complexity of matrix multiplication, relying on principles similar to Strassen's method: a reduction in the number of operations it takes to multiply sub-matrices of the original matrices to be multiplied. Coppersmith and Winograd (1987) identified an algorithm that has the asymptotic complexity of $O(n^{2.375477})$. Others have slightly improved that algorithm, and currently there is an algorithm for matrix multiplication with $M(n) = O(n^{\omega})$ such that $\omega = 2.3728639$ (Le Gall 2014). It is known that $M(n) = \Omega(n^2 \log n)$ (Raz 2002).

Although the asymptotically best matrix multiplication algorithms have large constant factors lurking in the O-notation, Strassen's algorithm does not, and is widely used in practice. Benedí and Sánchez (2007) show speed improvement when parsing natural language sentences using Strassen's algorithm as the matrix multiplication subroutine for Valiant's algorithm for CFG parsing. This indicates that similar speed-ups may be possible in practice using our algorithm for LCFRS parsing.

1.2 Main Result

Our main result is a matrix multiplication algorithm for **unbalanced**, **single-initial** binary LCFRS with asymptotic complexity $M(n^d) = O(n^{\omega d})$ where d is the maximal

---

1 Without placing a bound on f, the problem of recognition of LCFRS languages is NP-hard (Satta 1992).

number of combination points in all grammar rules. The constant d can be easily determined from the grammar at hand:

$$d = \max_{A \to BC} \max \left\{ \begin{array}{l} \varphi(A) + \varphi(B) - \varphi(C), \\ \varphi(A) - \varphi(B) + \varphi(C), \\ -\varphi(A) + \varphi(B) + \varphi(C) \end{array} \right\}$$

where $A \to BC$ ranges over rules in the grammar and $\varphi(A)$ is the fan-out of nonterminal A. Single-initial grammars are defined in Section 2, and include common formalisms such as tree-adjoining grammars. Any LCFRS can be converted to single-initial form by increasing its fan-out by at most one. The notion of unbalanced grammars is introduced in Section 4.4, and it is a condition on the set of LCFRS grammar rules that is satisfied with many practical grammars. In cases where the grammar is balanced, our algorithm can be used as a subroutine so that it parses the binary LCFRS in time $O(n^{\omega d+1})$. A similar procedure was applied by Nakanishi et al. (1998) for multiple component context-free grammars. See more discussion of this in Section 7.5.

Our results focus on the asymptotic complexity as a function of *string length*. We do not give explicit grammar constants. For other work that focuses on reducing the grammar constant in parsing, see, for example, Eisner and Satta (1999), Dunlop, Bodenstab, and Roark (2010), and Cohen, Satta, and Collins (2013). For a discussion of the optimality of the grammar constants in Valiant's algorithm, see, for example, Abboud, Backurs, and Williams (2015).

2. Background and Notation

This section provides background on LCFRS, and establishes notation used in the remainder of the paper. A reference table of notation is also provided in Appendix A.

For an integer n, let $[n]$ denote the set of integers $\{1, \ldots, n\}$. Let $[n]_0 = [n] \cup \{0\}$. For a set X, we denote by X^+ the set of all sequences of length 1 or more of elements from X.

A **span** is a pair of integers denoting left and right endpoints for a substring in a larger string. The endpoints are placed in the "spaces" between the symbols in a string. For example, the span $(0, 3)$ spans the first three symbols in the string. For a string of length n, the set of potential endpoints is $[n]_0$.

We turn now to give a succinct definition for binary LCFRS. For more details about LCFRS and their relationship to other grammar formalisms, see Kallmeyer (2010). A binary LCFRS is a tuple $(\mathcal{L}, \mathcal{T}, \mathcal{R}, \varphi, S)$ such that:

- $\mathcal{L}$ is the set of nonterminal symbols in the grammar.

- $\mathcal{T}$ is the set of terminal symbols in the grammar. We assume $\mathcal{L} \cap \mathcal{T} = \emptyset$.

- φ is a function specifying a fixed fan-out for each nonterminal ($\varphi \colon \mathcal{L} \to \mathbb{N}$).

- $\mathcal{R}$ is a set of productions. Each production p has the form $A \to g[B, C]$ where $A, B, C \in \mathcal{L}$, and g is a composition function $g \colon (\mathcal{T}^*)^{\varphi(B)} \times (\mathcal{T}^*)^{\varphi(C)} \to (\mathcal{T}^*)^{\varphi(A)}$, which specifies how to assemble the $\varphi(B) + \varphi(C)$ spans of the right-hand side nonterminals into the $\varphi(A)$ spans of the left-hand side nonterminal. We use square brackets as part of the syntax for writing productions, and parentheses to denote the application of the function g. The function g must be **linear** and

non-erasing, which means that if g is applied on a pair of tuples of strings, then each input string appears exactly once in the output, possibly as a *substring* of one of the strings in the output tuple. Rules may also take the form $A \to g[]$, where g returns a constant tuple of one string from $\mathcal{T}$.

- $S \in \mathcal{L}$ is a start symbol. Without loss of generality, we assume $\varphi(S) = 1$.

The language of an LCFRS $G = (\mathcal{L}, \mathcal{T}, \mathcal{R}, \varphi, S)$ is defined as follows:

- We define first the set yield(A) for every $A \in \mathcal{L}$:

  - For every $A \to g[] \in \mathcal{R}, g() \in$ yield(A).

  - For every $A \to g[B, C] \in \mathcal{R}$ and all tuples $\beta \in$ yield(B), $\gamma \in$ yield(C), $g(\beta, \gamma) \in$ yield(A).

  - Nothing else is in yield(A).

- The string language of G is $L(G) = \{w \mid \langle w \rangle \in$ yield(S)$\}$.

Intuitively, the process of generating a string from an LCFRS grammar consists of first choosing, top–down, a production to expand each nonterminal, and then, bottom–up, applying the composition functions associated with each production to build the string. As an example, the following context-free grammar:

$$S \to A\,B$$
$$A \to a$$
$$B \to b$$

corresponds to the following (binary) LCFRS:

$$S \to g_1[A, B] \qquad\qquad g_1(\langle \beta_1 \rangle, \langle \gamma_1 \rangle) = \langle \beta_1 \gamma_1 \rangle \qquad (1)$$
$$A \to g_2[] \qquad\qquad\qquad g_2() = \langle a \rangle$$
$$B \to g_3[] \qquad\qquad\qquad g_3() = \langle b \rangle$$

The only derivation possible under this grammar consists of the function application $g_1(g_2(), g_3()) = \langle ab \rangle$.

The following notation will be used to precisely represent the linear non-erasing composition functions g used in a specific grammar. For each production rule that operates on nonterminals A, B, and C, we define variables from the set $\mathcal{S} = \{\beta_1, \ldots, \beta_{\varphi(B)}, \gamma_1, \ldots, \gamma_{\varphi(C)}\}$. In addition, we define variables α_i for each rule where $i \in [\varphi(A)]$, taking values from $\mathcal{S}^+$. We write an LCFRS function as:

$$g(\langle \beta_1, \ldots, \beta_{\varphi(B)} \rangle, \langle \gamma_1, \ldots, \gamma_{\varphi(C)} \rangle) = \langle \alpha_1, \ldots, \alpha_{\varphi(A)} \rangle$$

where each $\alpha_i = \alpha_{i,1} \cdots \alpha_{i,n_i}$ specifies the parameter strings that are combined to form the ith string of the function's result tuple. For example, for the rule in Equation (1), $\alpha_{1,1} = \beta_1$ and $\alpha_{1,2} = \gamma_2$.

We adopt the following notational shorthand for LCFRS rules in the remainder of the article. We write the rule:

$$A \rightarrow g[B, C]$$
$$g(\langle \beta_1, \ldots, \beta_{\varphi(B)} \rangle, \langle \gamma_1, \ldots, \gamma_{\varphi(C)} \rangle) = \langle \alpha_1, \ldots, \alpha_{\varphi(A)} \rangle$$

as:

$$A[\alpha] \rightarrow B[\beta] \, C[\gamma]$$

where α consists of a tuple of strings from the alphabet $\{\beta_1, \ldots, \beta_{\varphi(B)}, \gamma_1, \ldots, \gamma_{\varphi(C)}\}$. In this notation, β is always the tuple $\langle \beta_1, \ldots, \beta_{\varphi(B)} \rangle$, and γ is always $\langle \gamma_1, \ldots, \gamma_{\varphi(C)} \rangle$. We include β and γ in the rule notation merely to remind the reader of the meaning of the symbols in α.

For example, with CFGs, rules have the form:

$$A[\langle \beta_1 \gamma_1 \rangle] \rightarrow B[\langle \beta_1 \rangle] \, C[\langle \gamma_1 \rangle]$$

indicating that B and C each have one span, and are concatenated in order to form A.

A binary TAG can also be represented as a binary LCFRS (Vijay-Shanker and Weir 1994). Figure 1 demonstrates how the adjunction operation is done with binary LCFRS. Each gray block denotes a span, and the adjunction operator takes the first span of nonterminal B and concatenates it to the first span of nonterminal C (to get the first span of A), and then takes the second span of C and concatenates it with the second span of B (to get the second span of A). For TAGs, rules have the form:

$$A[\langle \beta_1 \gamma_1, \gamma_2 \beta_2 \rangle] \rightarrow B[\langle \beta_1, \beta_2 \rangle] \, C[\langle \gamma_1, \gamma_2 \rangle] \tag{2}$$

The **fan-out** of a nonterminal is the number of spans in the input sentence that it covers. The fan-out of CFG rules is 1, and the fan-out of TAG rules is 2. The fan-out of the grammar, f, is the maximum fan-out of its nonterminals:

$$f = \max_{A \in \mathcal{L}} \varphi(A) \tag{3}$$

We sometimes refer to the **skeleton** of a grammar rule $A[\alpha] \rightarrow B[\beta] \, C[\gamma]$, which is just the context-free rule $A \rightarrow B \, C$, omitting the variables. In that context, a logical
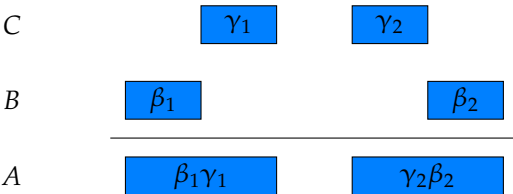

Figure 1
An example of a combination of spans for TAGs for the adjunction operation in terms of binary LCFRS. The rule in Equation (2) specifies how two nonterminals B and C are combined together into a nonterminal A.

statement such as $A \to B\,C \in \mathcal{R}$ is true if there is any rule $A[\alpha] \to B[\beta]\,C[\gamma] \in \mathcal{R}$ with some α, β, and γ.

For our parsing algorithm, we assume that the grammar is in a normal form such that the variables $\beta_1, \ldots, \beta_{\varphi(B)}$ appear in order in α, that is, that the spans of B are not re-ordered by the rule, and similarly we assume that $\gamma_1, \ldots, \gamma_{\varphi(C)}$ appear in order. If this is not the case in some rule, then the grammar can be transformed by introducing a new nonterminal for each permutation of a nonterminal that can be produced by the grammar. We further assume that $\alpha_{1,1} = \beta_1$, that is, that the first span of A begins with material produced by B rather than by C. If this not the case for some rule, B and C can be exchanged to satisfy this condition.

We refer to an LCFRS rule $A \to B\,C$ as **single-initial** if the leftmost endpoint of C is internal to a span of A, and **dual-initial** if the leftmost endpoint of C is the beginning of a span of A. Our algorithm will require the input LCFRS to be in single-initial form, meaning that all rules are single-initial. We note that grammars for common formalisms including TAG and synchronous context-free grammar (SCFG) are in this form. If a grammar is not in single-initial form, dual-initial rules can be converted to single-initial form by adding an empty span to B that combines with the first spans of C immediately to its left, as shown in Figure 2. Specifically, for each dual-initial rule $A \to B\,C$, if the first span of C appears between spans i and $i+1$ of B, create a new nonterminal B' with $\varphi(B') = \varphi(B) + 1$, and add a rule $B' \to B$, where B' produces B along with a span of length zero between spans i and $i+1$ of B. We then replace the rule $A \to B\,C$ with $A \to B'\,C$, where the new span of B' combines with C immediately to the left of C's first span. Because the new nonterminal B' has fan-out one greater than B, this grammar transformation can increase a grammar's fan-out by at most one.

By limiting ourselves to binary LCFRS grammars, we do not necessarily restrict the power of our results. Any LCFRS with arbitrary rank (i.e., with an arbitrary number of nonterminals in the right-hand side) can be converted to a binary LCFRS (with potentially a larger fan-out). See discussion in Section 7.6.

Example 1
Consider the phenomenon of cross-serial dependencies that exists in certain languages. It has been used in the past (Shieber 1985) to argue that Swiss–German is not context-free. One can show that there is a homomorphism between Swiss–German and the alphabet $\{a, b, c, d\}$ such that the image of the homomorphism intersected with the regular language $a^*b^*c^*d^*$ gives the language $L = \{a^m b^n c^m d^n \mid m, n \geq 1\}$. Because L is not context-free, this implies that Swiss-German is not context-free, because context-free languages are closed under intersection with regular languages.

Tree-adjoining grammars, on the other hand, are mildly context-sensitive formalisms that can handle such cross-serial dependencies in languages (where the *a*s
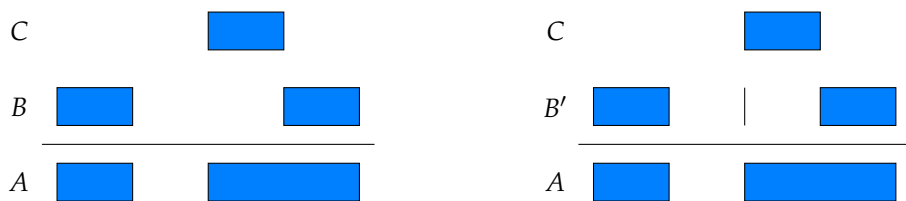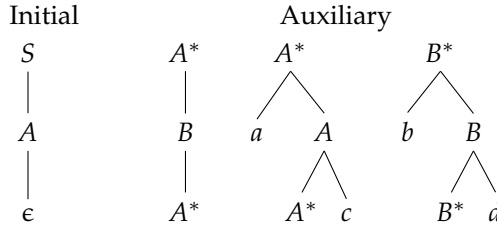

Figure 2
Conversion of a dual-initial rule to a single-initial rule.

are aligned with cs and the bs are aligned with the ds). For example, a tree-adjoining grammar for generating L would include the following initial and auxiliary trees (nodes marked by an asterisk are nodes where adjunction is not allowed):

Initial   Auxiliary

S A^* A^* B^*
| | /\   /\
A B a A b B
| | /\   /\
ϵ A^* A^* c B^* d

This TAG corresponds to the following LCFRS:

$$S \rightarrow g_1[A] \qquad\qquad g_1(\langle\beta_1, \beta_2\rangle) = \langle\beta_1\beta_2\rangle$$

$$A \rightarrow g_4[B] \qquad\qquad g_4(\langle\beta_1, \beta_2\rangle) = \langle\beta_1, \beta_2\rangle$$

$$A \rightarrow g_2[A] \qquad\qquad g_2(\langle\beta_1, \beta_2\rangle) = \langle a\beta_1, c\beta_2\rangle$$

$$B \rightarrow g_5[B] \qquad\qquad g_5(\langle\beta_1, \beta_2\rangle) = \langle b\beta_1, d\beta_2\rangle$$

$$A \rightarrow g_3[] \qquad\qquad g_3() = \langle\varepsilon, \varepsilon\rangle$$

$$B \rightarrow g_6[] \qquad\qquad g_6() = \langle\varepsilon, \varepsilon\rangle$$

Here we have one unary LCFRS rule for the initial tree, one unary rule for each adjunction tree, and one null-ary rule for each nonterminal producing a tuple of empty strings in order to represent TAG tree nodes at which no adjunction occurs. The LCFRS shown here does not satisfy our normal form requiring each rule to have either two nonterminals on the right-hand side with no terminals in the composition function, or zero nonterminals with a composition function returning fixed strings of terminals. However, it can be converted to such a form through a process analogous to converting a CFG to Chomsky Normal Form. For adjunction trees, the two strings returned by the composition function correspond the the material to the left and right of the foot node. The composition function merges terminals at the leaves of the adjunction tree with material produced by internal nodes of the tree at which adjunction may occur.

In general, binary LCFRS are more expressive than TAGs because they can have nonterminals with fan-out greater than 2, and because they can interleave the arguments of the composition function in any order.

3. A Sketch of the Algorithm

Our algorithm for LCFRS string recognition is inspired by the algorithm of Valiant (1975). It introduces a few important novelties that make it possible to use matrix multiplication for the goal of LCFRS recognition.

The algorithm relies on the observation that it is possible to construct a matrix T with a specific non-associative multiplication and addition operator such that multiplying T by itself k times on the left or on the right yields k-step derivations for a given string. The row and column indices of the matrix together assemble a set of spans in the string (the fan-out of the grammar determines the number of spans). Each cell in the matrix keeps track of the nonterminals that can dominate these spans. Therefore, computing the transitive closure of this matrix yields in each matrix cell the set of

nonterminals that can dominate the assembled indices' spans for the specific string at hand.

There are several key differences between Valiant's algorithm and our algorithm. Valiant's algorithm has a rather simple matrix-indexing scheme for the matrix: The rows correspond to the left endpoints of a span and the columns correspond to its right endpoints. Our matrix indexing scheme can mix both left endpoints and right endpoints at either the rows or the columns. This is necessary because with LCFRS, spans for the right-hand side of an LCFRS rule can combine in various ways into a new set of spans for the left-hand side.

In addition, our indexing scheme is "over-complete." This means that different cells in the matrix T (or its matrix powers) are equivalent and should consist of the same nonterminals. The reason we need such an over-complete scheme is again because of the possible ways spans of a right-hand side can combine in an LCFRS. To address this over-completeness, we introduce into the multiplication operator a "copy operation" that copies nonterminals between cells in order to maintain the same set of nonterminals in equivalent cells.

To give a preliminary example, consider the tree-adjoining grammar rule shown in Figure 1. We consider an application of the rule with the endpoints of each span instantiated as shown in Figure 3. With our algorithm, this operation will translate into the following sequence of matrix transformations. We will start with the following matrices, T_1 and T_2:

$$
T_1 \quad\quad (2,7) \quad\quad\quad\quad T_2 \quad\quad\quad (4,5)
$$

$$
(1,8)\begin{pmatrix} & \{\ldots,B,\ldots\} & \\ & & \\ & & \\ & & \end{pmatrix} \quad (2,7)\begin{pmatrix} & \{\ldots,C,\ldots\} & \\ & & \\ & & \\ & & \end{pmatrix}.
$$

For T_1, for example, the fact that B appears for the pair of addresses $(1,8)$ (for row) and $(2,7)$ for column denotes that B spans the constituents $(1,2)$ and $(7,8)$ in the string

$$
\begin{aligned}
&[C,j,k,\ell,m]\\
&[B,i,j,m,n]\\
\hline
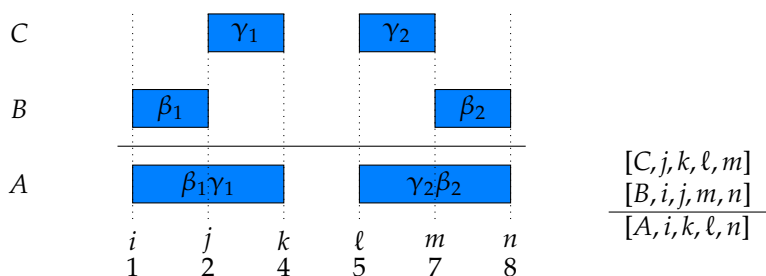&[A,i,k,\ell,n]
\end{aligned}
$$

Figure 3
A demonstration of a parsing step for the combination of spans in Figure 1. During parsing, the endpoints of each span are instantiated with indices into the string. The variables for these indices shown on the left correspond to the logical induction rule on the right. The specific choice of indices shown at the bottom is used in our matrix multiplication example in Section 3.

(this is assumed to be true—in practice, it is the result of a previous step of matrix multiplication). Similarly, with T_2, C spans the constituents $(2, 4)$ and $(5, 7)$.

Note that $(2, 7)$ are the two positions in the string where B and C meet, and that because B and C share these two endpoints, they can combine to form A. In the matrix representation, $(2, 7)$ appears as the column address of B and as the row address of C, meaning that B and C appear in cells that are combined during matrix multiplication. The result of multiplying T_1 by T_2 is the following:

$$T_1 T_2 \qquad\qquad (4, 5)$$

$$(1, 8) \begin{pmatrix} \{\ldots, A, \ldots\} \\[4em] \end{pmatrix}.$$

Now A appears in the cell that corresponds to the spans $(1, 4)$ and $(5, 8)$. This is the result of merging the spans $(1, 2)$ with $(2, 4)$ (left span of B and left span of C) into $(1, 4)$ and the merging of the spans $(5, 7)$ and $(7, 8)$ (right span of C and right span of B) into $(5, 8)$. Finally, an additional copying operation will lead to the following matrix:

$$T_3 \qquad\qquad (5, 8)$$

$$(1, 4) \begin{pmatrix} \{\ldots, A, \ldots\} \\[4em] \end{pmatrix}.$$

Here, we copy the nonterminal A from the address with the row $(1, 8)$ and column $(4, 5)$ into the address with the row $(1, 4)$ and column $(5, 8)$. Both of these addresses correspond to the same spans $(1, 4)$ and $(5, 8)$. Note that matrix row and column addresses can mix both starting points of spans and ending points of spans.

4. A Matrix Multiplication Algorithm for LCFRS

We turn next to give a description of the algorithm. Our description is constructed as follows:

- In Section 4.1 we describe the basic matrix structure used for LCFRS recognition. This construction depends on a parameter d, the contact rank, which is a function of the underlying LCFRS grammar we parse with. We also describe how to create a seed matrix, for which we need to compute the transitive closure.

- In Section 4.2 we define the multiplication operator between cells of the matrices we use. This multiplication operator is distributive, but not associative, and as such we use Valiant's specialized transitive closure algorithm to compute transitive closure of the seed matrix given a string.

- In Section 4.3 we define the contact rank parameter d. The smaller d is, the more efficient it is to parse with the specific grammar.

- In Section 4.4 we define when a binary LCFRS is "balanced." This is an end case that increases the final complexity of our algorithm by a factor of $O(n)$. Nevertheless, it is an important end case that appears in applications, such as inversion transduction grammars.

- In Section 4.5 we tie things together, and show that computing the transitive closure of the seed matrix we define in Section 4.1 yields a recognition algorithm for LCFRS.

4.1 Matrix Structure

The algorithm will seek to compute the transitive closure of a seed matrix $T(d)$, where d is a constant determined by the grammar (see Section 4.3). The matrix rows and columns are indexed by the set $N(d)$ defined as:

$$N(d) = \bigcup_{d'=1}^{d} ([n]_0 \times \{0,1\})^{d'} \tag{4}$$

where n denotes the length of the sentence, and the exponent d' denotes a repeated Cartesian product. Thus each element of $N(d)$ is a sequence of indices into the string, where each index is annotated with a bit (an element of the set $\{0,1\}$) indicating whether it is **marked** or unmarked. Marked indices will be used in the copy operator defined later. Indices are unmarked unless specified as marked: We use $\hat{x}$ to denote a marked index $(x, 1)$ with $x \in [n]_0$.

In the following, it will be safe to assume sequences from $N(d)$ are monotonically increasing in their indices. For an $i \in N(d)$, we overload notation, and often refer to the set of all elements in the first coordinate of each element in the sequence (ignoring the additional bits). As such,

- The set $i \cup j$ is defined for $j \in N(d)$.

- If we state that i is in $N(d)$ and includes a *set* of endpoints, it means that i is a sequence of these integers (ordered lexicographically) with the bit part determined as explained in the context (for example, all unmarked).

- The quantity $|i|$ denotes the length of the sequence.

- The quantity $\min i$ denotes the smallest index among the first coordinates of all elements in the sequence i (ignoring the additional bits).

We emphasize that the variables i, j, and k are mostly elements in $N(d)$ as overloaded above, not integers, throughout this article; we choose the symbols i, j, and

k by analogy to the variables in the CKY parsing algorithm, and also because we use the sequences as addresses for matrix rows and columns. For $i, j \in N(d)$, we define $m(i,j)$ to be the set of $f' = \frac{1}{2}|i \cup j|$ pairs $\{(\ell_1, \ell_2), (\ell_3, \ell_4), \ldots, (\ell_{2f'-1}, \ell_{2f'})\}$ such that $\ell_k < \ell_{k+1}$ for $k \in [2f'-1]$ and $(\ell_k, 0) \in i \cup j$ for $k \in [2f']$. This means that $m(i,j)$ takes as input the two sequences in matrix indices, merges them, sorts them, then divides this sorted list into a set of f' consecutive pairs. Whenever $\min j \le \min i$, $m(i,j)$ is undefined. The interpretation of this is that ℓ_1 should always belong to i and not j. See more details in Section 4.2. In addition, if any element of i or j is marked, $m(i,j)$ is undefined.

We define an order $<$ on elements i and j of $N(d)$ by first sorting the sequences i and j and then comparing i and j lexicographically (ignoring the bits). This ensures that $i < j$ if $\min i < \min j$. We assume that the rows and columns of our matrices are arranged in this order. For the rest of the discussion, we assume that d is a constant, and refer to $T(d)$ as T and $N(d)$ as N.

We also define the set of triples M as the following Cartesian product:

$$M = (\mathcal{L} \cup \{\text{⊴},\text{⬠▸},\text{⊵},\text{⬠▸},\text{▭},\text{▯}\}) \times N \times N$$

where ⊴, ⬠▸, ⬠▸, ⊵, ▭, and ▯ are six special pre-defined symbols.[2] Each cell T_{ij} in T is a set such that $T_{ij} \subset M$.

The intuition behind matrices of the type of T (meaning T and, as we see later, products of T with itself, or its transitive closure) is that each cell indexed by (i,j) in such a matrix consists of all nonterminals that can be generated by the grammar when parsing a sentence such that these nonterminals span the constituents $m(i,j)$ (whenever $m(i,j)$ is defined). Our normal form for LCFRS ensures that spans of a nonterminal are never re-ordered, meaning that it is not necessary to retain information about which indices demarcate which components of the nonterminal, because one can sort the indices and take the first two indices as delimiting the first span, the second two indices as delimiting the second span, and so on. The two additional N elements in each triplet in a cell are actually just copies of the row and column indices of that cell. As such, they are identical for all triplets in that cell. The additional ⊴, ⬠▸, ⬠▸, ⊵, ▭, ▯ symbols are symbols that indicate to the matrix multiplication operator that a "copying operation" should happen between equivalent cells (Section 4.2).

Figure 4 gives an algorithm to seed the initial matrix T. Entries added in Step 2 of the algorithm correspond to entries in the LCFRS parsing chart that can be derived immediately from terminals in the string. Entries added in Step 3 of the algorithm do not depend on the input string or input grammar, but rather initialize elements used in the copy operation described in detail in Section 4.2. Because the algorithm only initializes entries with $i < j$, the matrix T is guaranteed to be upper triangular, a fact which we will take advantage of in Section 4.2.

4.1.1 Configurations. Our matrix representation requires that a nonterminal appears in more than one equivalent cell in the matrix, and the specific set of cells required depends on the specific patterns in which spans are combined in the LCFRS

---

2 The symbols will be used for "copying commands:" (1) "from row" (⊴); (2) "from column" (⬠▸); (3) "to row" (⊵); (4) "to column" (⬠▸); (5) "unmark row" (▭); (6) "unmark column" (▯).

Inputs: An LCFRS grammar as defined in §2 and a sentence $w_1 \cdots w_n$.

Outputs: A seed matrix T with rows and columns indexed by N, such that each cell in T is a subset of M

Algorithm:

1. Set $T_{ij} = \emptyset$ for all $i, j \in N$.
2. For each $i, j \in N$, for each nonterminal $A \in \mathcal{L}$, set $T_{ij} \leftarrow T_{ij} \cup \{(A, i, j)\}$ if $m(i, j) = \{(\ell_1, \ell_2), (\ell_3, \ell_4), \ldots, (\ell_{2f-1}, \ell_{2f})\}$ and there is a rule in the grammar $A \to g(), g() = \langle \alpha_1, \ldots, \alpha_{\varphi(A)} \rangle$, where $\alpha_i = \alpha_{i,1} \cdots \alpha_{i,n_i}$, and for each i and j, $\alpha_{i,j} = w_{\ell_{2i-1}+j}$ (i.e., $\alpha_{i,j}$ is the $(\ell_{2i-1} + j)$th word in the sentence).
3. For each $i, j \in N$ such that $i < j$
   a. $T_{ij} \leftarrow T_{ij} \cup \{(\square\!\!\text{+}, i, j)\}$ if all indices in i are unmarked and $j = \mathrm{insert}(i, \hat{x})$ for some x
   b. $T_{ij} \leftarrow T_{ij} \cup \{(\sqcup\!\!\!\perp, i, j)\}$ if all indices in j are unmarked and $i = \mathrm{remove}(j, x)$ for some x
   c. $T_{ij} \leftarrow T_{ij} \cup \{(\square, i, j)\}$ if all indices in i are unmarked and $i = \mathrm{insert}(\mathrm{remove}(j, \hat{x}), x)$ for some x
   d. $T_{ij} \leftarrow T_{ij} \cup \{(\sqcup\!\!\!\perp, i, j)\}$ if all indices in j are unmarked and $i = \mathrm{insert}(j, \hat{x})$ for some x
   e. $T_{ij} \leftarrow T_{ij} \cup \{(\square\!\!\text{+}, i, j)\}$ if all indices in i are unmarked and $j = \mathrm{remove}(i, x)$ for some x
   f. $T_{ij} \leftarrow T_{ij} \cup \{(\square, i, j)\}$ if all indices in j are unmarked and $j = \mathrm{insert}(\mathrm{remove}(i, \hat{x}), x)$ for some x

Figure 4
An algorithm for computing the seed matrix T. The function $\mathrm{remove}(v, x)$ takes a sequence of integers v and removes x from it, if it is in there. The function $\mathrm{insert}(v, x)$ takes a sequence of integers and adds x to it.

grammar. We now present a precise description of these cells by defining the **configuration** of a nonterminal in a rule. The concept of a configuration is designed to represent which endpoints of spans of the rule's right-hand side (r.h.s.) nonterminals B and C meet one another to produce larger spans, and which endpoints, on the other hand, become endpoints of spans of the left-hand side (l.h.s.) nonterminal A.

For each of the three nonterminals involved in a rule, the configuration is the set of endpoints in the row address of the nonterminal's matrix cell. To make this precise, for a nonterminal B with fan-out $\varphi(B)$, we number the endpoints of spans with integers in the range 1 to $2\varphi(B)$. In a rule $A[\alpha] \to B[\beta]\, C[\gamma]$, the configuration of B is the subset of $[2\varphi(B)]$ of endpoints of B that do not combine with endpoints of C in order to form a single span of A. The endpoints will form the row address for B. Formally, let $\beta = \langle \beta_1, \ldots, \beta_{\varphi(B)} \rangle$, and let $\alpha = \langle \alpha_{1,1} \cdots \alpha_{1,n_1}, \ldots, \alpha_{\varphi(A),1} \cdots \alpha_{\varphi(A),n_{\varphi(A)}} \rangle$. Then the set of non-combination endpoints of B is defined as:

$$\mathrm{config}_2(r) = \{2i \mid \beta_i = \alpha_{j,n_j} \text{ for some } j\} \cup \{2i - 1 \mid \beta_i = \alpha_{j,1} \text{ for some } j\}$$

where the first set defines right ends of spans of B that are right ends of some span of A, and the second set defines left ends of spans of B that are left ends of some span of A. For example, given that CFG rules have the form

$$r = A[\langle \beta_1 \gamma_1 \rangle] \rightarrow B[\langle \beta_1 \rangle] \, C[\langle \gamma_1 \rangle]$$

the configuration $\mathrm{config}_2(r)$ is $\{1\}$ because, of B's two endpoints, only the first is also an endpoint of A. For the TAG rule t shown in Figure 1, $\mathrm{config}_2(t) = \{1, 4\}$ because, of B's four endpoints, the first and fourth are also endpoints of A.

For the second r.h.s. nonterminal of a rule r, the configuration consists of the set of endpoints in the row address for C, which are the endpoints that *do* combine with B:

$$\mathrm{config}_3(r) = \{2i \mid \gamma_i = \alpha_{j,k} \text{ for some } 1 \leq k < n_j\} \cup \{2i - 1 \mid \gamma_i = \alpha_{j,k} \text{ for some } 1 < k \leq n_j\}$$

where the first set defines right ends of spans of C that are internal to some span of A, and the second set defines left ends of spans of C that are internal to some span of A. For example, any CFG rule r has configuration, $\mathrm{config}_3(r) = \{1\}$, because the first endpoint of C is internal to A. For the TAG rule t shown in Figure 1, $\mathrm{config}_3(t) = \{1, 4\}$ because, of C's four endpoints, the first and fourth are internal A.

For the l.h.s. nonterminal A of the rule, matrix multiplication will produce an entry in the matrix cell where the row address corresponds to the endpoints from B, and the column address corresponds to the endpoints from C. To capture this partition of the endpoints of A, we define

$$\mathrm{config}_1(r) = \{2i \mid \alpha_{i,n_i} = \beta_j \text{ for some } j\} \cup \{2i - 1 \mid \alpha_{i,1} = \beta_j \text{ for some } j\}$$

where the first set defines right ends of spans of A that are formed from B, and the second set defines left ends of spans of A that are formed from B. For example, any CFG rule r has configuration, $\mathrm{config}_1(r) = \{1\}$, because only the first endpoint of A is derived from B. For the TAG rule t shown in Figure 1, $\mathrm{config}_1(t) = \{1, 4\}$ because, of A's four endpoints, the first and fourth are derived from B.

4.2 Definition of Multiplication Operator

We need to define a multiplication operator $\otimes$ between a pair of elements $R, S \subset M$. Such a multiplication operator induces multiplication between matrices of the type of T, just by defining for two such matrices, T_1 and T_2, a new matrix of the same size $T_1 \otimes T_2$ such that:

$$[T_1 \otimes T_2]_{ij} = \bigcup_{k \in N} \left([T_1]_{ik} \otimes [T_2]_{kj} \right) \tag{5}$$

We also use the $\cup$ symbol to denote coordinate-wise union of cells in the matrices it operates on.

The operator $\otimes$ we define is not associative, but it is distributive over $\cup$. This means that for $R, S_1, S_2 \subset M$ it holds that:

$$R \otimes (S_1 \cup S_2) = (R \otimes S_1) \cup (R \otimes S_2)$$

In addition, whenever $R = \emptyset$, then for any S, $R \otimes S = S \otimes R = \emptyset$. This property maintains the upper-triangularity of the transitive closure of T.

Figure 5 gives the algorithm for multiplying two elements of the matrix. The algorithm is composed of two components. The first component (Step 2 in Figure 5) adds nonterminals, for example, A, to cell (i, j), if there is some B and C in (i, k) and (k, j), respectively, such that there exists a rule $A \to B\,C$ *and* the span endpoints denoted by k are the points where the rule specifies that spans of B and C should meet.

In order to make this first component valid, we have to make sure that k can indeed serve as a concatenation point for (i, j). Step 2 verifies this using the concept of configurations defined earlier. To apply a rule $r : A[\alpha] \to B[\beta]\,C[\gamma]$, we must have an entry for (B, i, k) in cell (i, k), where i is a set of indices corresponding to the endpoints of B selected by $\text{config}_2(r)$ and k is a set of indices corresponding to the endpoints of B selected by $[2\varphi(B)] \setminus \text{config}_2(r)$. This condition is enforced by Step 2c of Figure 5. Similarly, we must have an entry for (C, k, j) in cell (k, j), where k is a set of indices corresponding to the endpoints of C selected by $\text{config}_3(r)$ and j is a set of indices corresponding to the endpoints of C selected by $[2\varphi(C)] \setminus \text{config}_3(r)$. This is enforced by Step 2d. Finally, the spans defined by B and C must not overlap in the string. To

---

Inputs: A pair of elements $R, S \subset M$.
Outputs: A new subset of M, denoted by $(R \otimes S)$.
Algorithm:

1. $(R \otimes S) = \emptyset$
2. For each pair of elements $r = (B, i, k)$ and $s = (C, k, j)$ where $r \in R$ and $s \in S$ add an element (A, i, j) to $(R \otimes S)$ if:
   a. There is a binary rule in the LCFRS grammar $r = A[\alpha] \to B[\beta]\,C[\gamma]$
   b. All indices in i, j, and k are unmarked
   c. Configuration of B: assume $m(i, k) = \{(\ell_1, \ell_2), \ldots, (\ell_{2\varphi(B)-1}, \ell_{2\varphi(B)})\}$
      We require that $i = \{\ell_q \mid q \in \text{config}_2(r)\}$
   d. Configuration of C: assume $m(k, j) = \{(\ell'_1, \ell'_2), \ldots, (\ell'_{2\varphi(C)-1}, \ell'_{2\varphi(C)})\}$
      We require that $k = \{\ell'_q \mid q \in \text{config}_3(r)\}$
   e. Configuration of A: assume $m(i, j) = \{(\ell''_1, \ell''_2), \ldots, (\ell''_{2\varphi(A)-1}, \ell''_{2\varphi(A)})\}$
      We require that $i = \{\ell''_q \mid q \in \text{config}_1(r)\}$
3. For each pair of elements $r = (A, i, k)$ and $s = (\boxminus^\leftarrow, k, j)$, add the element (A, i, j) to $(R \otimes S)$ if: i contains only unmarked indices, and $\hat{x} \in j$ and $x \in i$ for some x
4. For each pair of elements $r = (\boxminus, i, k)$ and $s = (A, k, j)$, add the element (A, i, j) to $(R \otimes S)$ if: $\hat{x} \in j$ and $x \notin i$ and for some x
5. For each pair of elements $r = (A, i, k)$ and $s = (\boxminus, k, j)$, add the element (A, i, j) to $(R \otimes S)$ if: $|i \cup j| = 2\varphi(A)$
6. For each pair of elements $r = (\boxminus, i, k)$ and $s = (A, k, j)$, add the element (A, i, j) to $(R \otimes S)$ if: j contains only unmarked indices, and $\hat{x} \in i$ and $x \in j$ for some x
7. For each pair of elements $r = (A, i, k)$ and $s = (\boxminus^\rightarrow, k, j)$, add the element (A, i, j) to $(R \otimes S)$ if: $\hat{x} \in i$ and $x \notin j$ for some x
8. For each pair of elements $r = (\boxminus, i, k)$ and $s = (A, k, j)$, add the element (A, i, j) to $(R \otimes S)$ if: $|i \cup j| = 2\varphi(A)$

Figure 5
An algorithm for the product of two matrix elements.

guarantee that the spans do not overlap, we sort the endpoints of A and check that each position in the sorted list is derived from either B or C as required by the configuration of A in r. This check is performed in Step 2e of Figure 5.

Given that T is initialized to be upper-triangular, the properties of matrix multiplication guarantee that all matrix powers of T are upper-triangular. We now proceed to show that upper-triangular matrices are sufficient in terms of the grammar. In particular, we need to show the following lemma:

Lemma 1
For each application of a single-initial rule $A \rightarrow B\,C$, it is possible to create an entry for A by multiplying two upper-triangular matrices T_1 and T_2, where T_1 contains an entry for B, and T_2 contains an entry for C.

Proof
A nonterminal B appears in a cell above the diagonal if its row address is smaller than its column address, which in turn occurs if the leftmost endpoint of B appears in the row address rather than the column address. The row address for B contains the endpoints of B that are also endpoints of A. Our normal form for LCFRS rules ensures that the leftmost endpoint of B forms the leftmost endpoint of A. Therefore the leftmost endpoint of B is in B's row address, and B is above the diagonal.

The row address of nonterminal C in T_2 must contain the endpoints of C that combine with endpoints of B. For single-initial rules, these endpoints include the leftmost endpoint of C, guaranteeing that C appears above the diagonal.

Because each instance of A can be produced by combining elements of T_1 and T_2 that are above the diagonal, each instance of A can be produced by multiplying two upper-triangular matrices. ∎

4.2.1 Copy Operations. The first component of the algorithm is sound, but not complete. If we were to use just this component in the algorithm, then we would obtain in each cell (i, j) of the transitive closure of T a *subset* of the possible nonterminals that can span $m(i, j)$. The reason this happens is that our addressing scheme is "over-complete." This means that any pair of addresses (i, j) and (k, ℓ) are equivalent if $m(i, j) = m(k, \ell)$.

We need to ensure that the transitive closure, using $\otimes$, propagates, or copies, nonterminals from one cell to its equivalents. This is done by the second component of the algorithm, in Steps 3–6. The algorithm does this kind of copying by using a set of six special "copy" symbols, $\{ ⧨, ⧩, ⧨, ⧩, ◫, ⬚ \}$. These symbols copy nonterminals from one cell to the other in multiple stages.

Suppose that we need to copy a nonterminal from cell (i, j) to cell (k, ℓ), where $m(i, j) = m(k, \ell)$, indicating that the two cells describe the same set of indices in the input string. We must move the indices in $i \cap \ell$ from the row address to the column address, and we must move the indices in $j \cap k$ from the column address to the row address. We will move one index at a time, adding nonterminals to intermediate cells along the way.

We now illustrate how our operations move a single index from a row address to a column address (moving from column to row is similar). Let x indicate the index we wish to move, meaning that we wish to copy a nonterminal in cell (i, j) to cell $(\text{remove}(i, x), \text{insert}(j, x))$. Because we want our overall parsing algorithm to take advantage of fast matrix multiplication, we accomplish the copy operations through a sequence of three matrix multiplications, as shown in Figure 6. The first multiplication involves the nonterminal A in cell (i, j) in the left matrix, and a ⧨ symbol in cell $(j, \text{insert}(j, \hat{x}))$ in the right matrix, resulting in a matrix with nonterminal A in cell
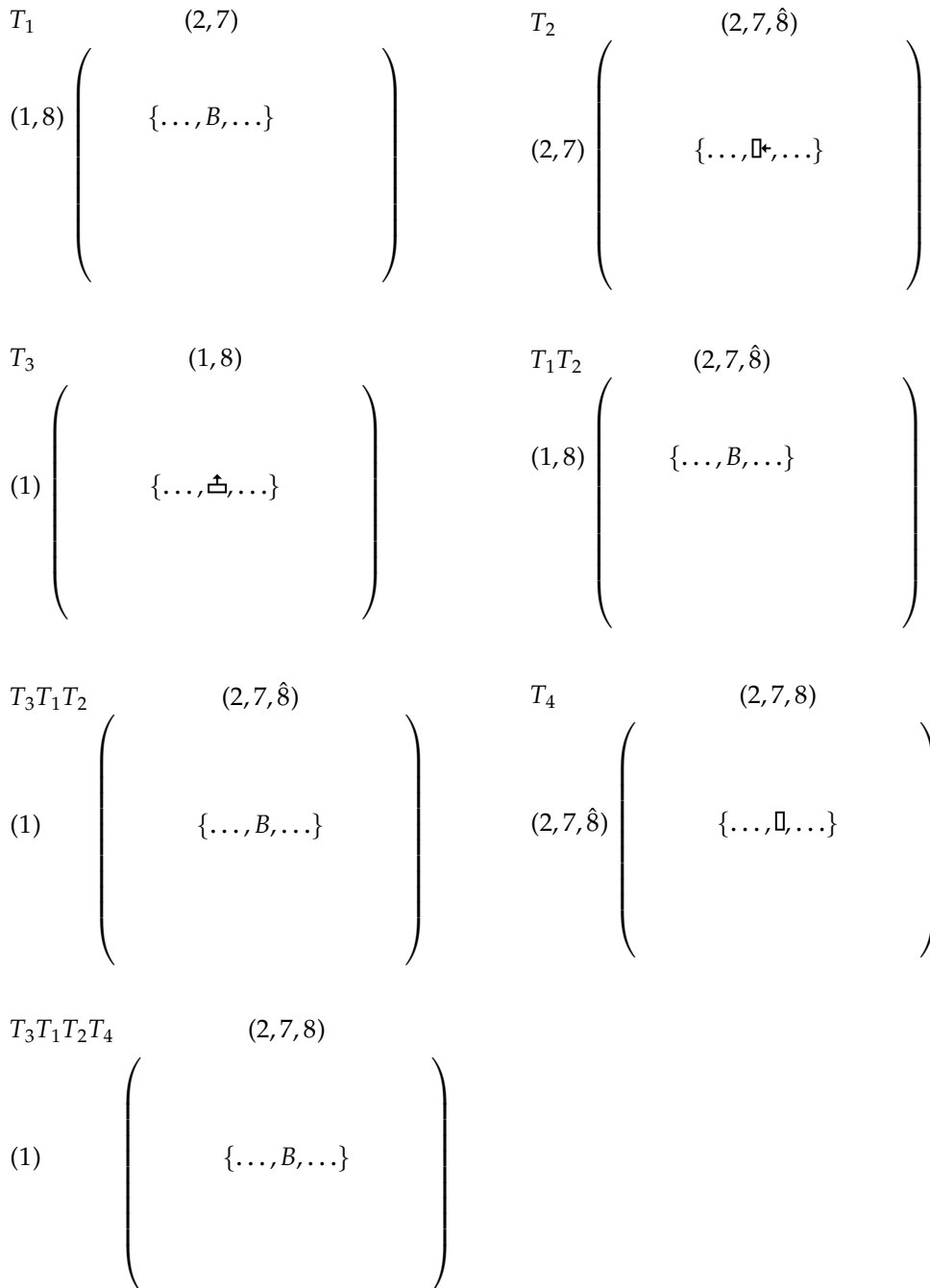
T_1 $(2,7)$

$(1,8)$ $\begin{pmatrix} \{\ldots, B, \ldots\} \end{pmatrix}$

T_2 $(2,7,\hat{8})$

$(2,7)$ $\begin{pmatrix} \{\ldots, \square\!\leftarrow, \ldots\} \end{pmatrix}$

T_3 $(1,8)$

(1) $\begin{pmatrix} \{\ldots, \square, \ldots\} \end{pmatrix}$

$T_1 T_2$ $(2,7,\hat{8})$

$(1,8)$ $\begin{pmatrix} \{\ldots, B, \ldots\} \end{pmatrix}$

$T_3 T_1 T_2$ $(2,7,\hat{8})$

(1) $\begin{pmatrix} \{\ldots, B, \ldots\} \end{pmatrix}$

T_4 $(2,7,8)$

$(2,7,\hat{8})$ $\begin{pmatrix} \{\ldots, \square, \ldots\} \end{pmatrix}$

$T_3 T_1 T_2 T_4$ $(2,7,8)$

(1) $\begin{pmatrix} \{\ldots, B, \ldots\} \end{pmatrix}$

Figure 6
An example of moving an index from the row address to the column address. Nonterminal B in T_1 is copied from cell $(1,8), (2,7)$ to cell $(1), (2,7,8)$ through three matrix multiplications. First, multiplying by T_2 on the right yields $T_1 T_2$, shown in the right of the second row. Multiplying this matrix by T_3 on the left yields $T_1 T_2 T_3$. Finally, multiplying this matrix by T_4 on the right yields $T_1 T_2 T_3 T_4$, shown in the bottom row.

$(i, \text{insert}(j, \hat{x}))$. This intermediate result is redundant in the sense that index x appears in the row and index $\hat{x}$ appears in the column address. To remove x from the row address, we multiply on the left with a matrix containing the symbol ⛝ in cell $(\text{remove}(i, x), i)$, resulting in a matrix with nonterminal A in cell $(\text{remove}(i, x), \text{insert}(j, \hat{x}))$. Finally, we multiply by a third matrix to replace the marked index $\hat{x}$ with the unmarked index x. This is done by multiplying on the right with a matrix containing the symbol ⛝ in cell $(\text{insert}(j, \hat{x}), \text{insert}(j, x))$.

The key idea behind this three-step process is to copy elements from one cell to another through intermediate cells. In matrix multiplication, only cells that share a row or a column index actually interact when doing multiplication. Therefore, in order to copy a nonterminal from (i, j) to another cell which represents the same set of spans, we have to copy it through cells such as $(i, \text{insert}(j, \hat{x}))$ that share the index i with (i, j).

In order to guarantee that our operations copy nonterminals only into cells with equivalent addresses, the seed matrix contains the special symbol ⛝ only in cells (j, k) such that $k = \text{insert}(j, \hat{x})$ for some x. When ⛝ in cell (j, k) combines with a nonterminal A in cell (i, j), the result contains A only if $x \in i$, guaranteeing that the index added to the column address was originally present in the row address. In addition, the condition that i contains only unmarked indices (in the multiplication operator) and that the condition j contains only unmarked indices (in the initialization of the seed matrix) guarantee that only one index is marked in the address of any non-empty matrix cell.

Similar conditions apply to the ⛝ operation. The seed matrix contains ⛝ only in cells (i, k) such that $i = \text{remove}(k, x)$ for some x, guaranteeing that the operation only removes one index at a time. Furthermore, when ⛝ in cell (i, k) combines with a nonterminal A in cell (k, j), the result contains A only if $\hat{x} \in j$. This guarantees that the new entry includes all the original indices, meaning that any index we remove from the row address is still present as a marked index in the column address.

The ⛝ operator removes the mark on index $\hat{x}$ in the column address, completing the entire copying process. The condition $|i \cup j| = \varphi(A)$ ensures that the removal of the mark from $\hat{x}$ does not take place until after x has been removed from the row address.

Taken together, these conditions ensure that after a sequence of one ⛝, one ⛝, and one ⛝, A is copied into all cells having the form $(\text{remove}(i, x), \text{insert}(j, x))$ for some x.

To move an index from the column address to the row address, we use one ⛝ operation followed by one ⛝ operation and one ⛝ operation. The conditions on these three special symbols are analogous to the conditions on ⛝, ⛝, and ⛝ outlined earlier, and ensure that we copy from cell (i, j) to cells of the form $(\text{insert}(i, x), \text{remove}(j, x))$ for some x.

We now show that matrix powers of the upper-triangular seed matrix T copy nonterminals between all equivalent cells above the diagonal.

Lemma 2

Let (i, j) and (k, ℓ) be unmarked matrix addresses, in a seed matrix T indexed by row and column addresses from $N(d)$ where $d > \min\{|i|, |j|\}$ and $d > \min\{|k|, |\ell|\}$. Assume that $\min i = \min k$ and either $k = \text{remove}(i, x)$ and $\ell = \text{insert}(j, x)$ for some x, or $k = \text{insert}(i, x)$ and $\ell = \text{remove}(j, x)$ for some x. If A appears in cell (i, j) of $T^{(n)}$, then A appears in cell (k, ℓ) of $T^{(n+3)}$. Furthermore, the copy operations do not introduce nonterminals into any other cells with unmarked addresses.

Proof

The condition on d guarantees that we can form row and column addresses long enough to hold the redundant representations with one address shared between row

and column. This condition is only relevant in the case where i, j, k, and ℓ are all of the same length; in this case we need to construct temporary indices with length one greater, as in the example in Figure 6.

A can be added to cell (k, ℓ) through a sequence of three matrix multiplications by combining with symbols ⟐, ⟑, and ⟎ or with ⟑, ⟐, and ⟏. Because $T^{(n)}$ is upper triangular, $\min i = \min i \cup j$, meaning that A's leftmost index is in its row address. The condition $\min i = \min k$ implies that we are not moving this leftmost index from row to column. The addresses of the three copy symbols required are all formed by adding or removing x or $\hat{x}$ to the row and column addresses (i, j); because the leftmost index of i is not modified, the copy symbols that are required are all above the diagonal, and are present in the seed matrix T. Therefore, A appears in cell (k, ℓ) of $T^{(n+3)}$.

To see that nonterminals are not introduced into any other cells, observe that ⟎ and ⟏ are the only symbols that introduce nonterminals into unmarked addresses. They can only apply when a marked index is present, and when the total number indices is $2\varphi(A)$. This can only occur after either ⟐ has introduced a marked index and ⟑ removed the corresponding unmarked index, or ⟑ has introduced a marked index and ⟐ removed the corresponding unmarked index. ∎

Putting together sequences of these operations to move indices, we arrive at the following lemma:

Lemma 3
Let (i, j) and (k, ℓ) be matrix addresses such that $m(i, j) = m(k, \ell)$, in a seed matrix T indexed by row and column addresses from $N(d)$ where $d > \min\{|i|, |j|\}$ and $d > \min\{|k|, |\ell|\}$. Then, for any nonterminal A in cell (i, j) in $T^{(n)}$, A will also appear in cell (k, ℓ) of the power matrix $T^{(n+6d)}$.

Proof
Nonterminal A can be copied through a series of intermediate cells by moving one index at a time from i to ℓ, and from j to k. We begin by moving indices from either the row address i to the column address if $|i| > |j|$, or from the column address j to the row address otherwise. We must move up to d indices from row to column, and d indices from column to row. Each move takes three matrix multiplications, for a total of $6d$ matrix multiplications. ∎

4.3 Determining the Contact Rank

The dimensions of the matrix T (and its transitive closure) are $|N| \times |N|$. The set N is of size $O(n^d)$, where d is a function of the grammar. When a given pair of cells in two matrices of the type of T are multiplied, we are essentially combining endpoints from the first multiplicand column address with endpoints from the second multiplicand row address. As such, we have to ensure that d allows us to generate all possible sequences of endpoints that could potentially combine with a given fixed LCFRS.

We refer to the endpoints at which a rule's r.h.s. nonterminals meet as **combining points**. For example, in the simple case of a CFG with a rule $S \rightarrow NP\,VP$, there is one combining point where NP and VP meet. For the TAG rule shown in Figure 1, there are two combining points where nonterminals B and C meet. For each rule r in the LCFRS grammar, we must be able to access the combining points as row and column addresses in order to apply the rule with matrix multiplication. Thus, d must be at least

the maximum number of combining points of any rule in the grammar. The number of combining points $\delta(r)$ for a rule r can be computed by comparing the number of spans on the l.h.s. and r.h.s. of the rule:

$$\delta(A[\alpha] \to B[\beta]\,C[\gamma]) = \varphi(C) + \varphi(B) - \varphi(A) \tag{6}$$

Note that $\delta(r)$ depends only on the skeleton of r (see Section 2), and therefore it can be denoted by $\delta(A \to B\,C)$.[3]

For each nonterminal on the r.h.s. of the rule, the address of its matrix cell consists of the combination points in one dimension (either row or column), and the other points in the other dimension of the matrix. For r.h.s. nonterminal B in rule $A \to B\,C$, the number of non-combination endpoints is:

$$2\varphi(B) - \delta(A \to B\,C)$$

Thus, taking the maximum size over all addresses in the grammar, the largest addresses needed are of length:

$$d = \max_{A \to B\,C \in \mathcal{R}} \max \left\{ \begin{array}{l} \delta(A \to B\,C), \\ 2\varphi(B) - \delta(A \to B\,C), \\ 2\varphi(C) - \delta(A \to B\,C) \end{array} \right\}$$

We call this number the **contact rank** of the grammar. As examples, the contact rank of a CFG is 1, while the contact rank of a TAG is 2. A simple algebraic manipulation shows that the contact rank can be expressed as follows:

$$d = \max_{A \to B\,C \in \mathcal{R}} \max \left\{ \begin{array}{l} \varphi(A) + \varphi(B) - \varphi(C), \\ \varphi(A) - \varphi(B) + \varphi(C), \\ -\varphi(A) + \varphi(B) + \varphi(C) \end{array} \right\}$$

We require our grammars to be in single-initial form, as described in Section 2. Because the process of converting an LCFRS grammar to single-initial form increases its fan-out by at most one, the contact rank is also increased by at most one.

4.4 Balanced Grammars

We define the **configuration set** of a nonterminal A to the the set of all configurations (Section 4.1.1) in which A appears in a grammar rule, including both appearances in the

---

3 To see that Equation (6) is true, consider that if we take $\varphi(B) + \varphi(C)$ variables from the spans of the r.h.s. and try to combine them together to $\varphi(A)$ sequences per span of the l.h.s., we will get $\varphi(B) + \varphi(C) - \varphi(A)$ points where variables "touch." If $\varphi(A) = 1$, then this is clearly true. For $\varphi(A) > 1$, consider that for each span, we "lose" one contact point.

r.h.s. and as the l.h.s.

$$\text{config}(a) = \left(\bigcup_{r:\text{lhs}(r)=A} \{\text{config}_1(r)\} \right) \cup \left(\bigcup_{r:\text{rhs1}(r)=A} \{\text{config}_2(r)\} \right) \cup \left(\bigcup_{r:\text{rhs2}(r)=A} \{\text{config}_3(r)\} \right)$$

For example, in a CFG, the configuration set of any nonterminal is $\{\{1\}\}$, because, as shown in Section 4.1.1, nonterminals are always used in the unique configuration $\{1\}$. For TAG, the configuration set of any nonterminal is $\{\{1,4\}\}$ because, as in CFG, nonterminals are always used in the same configuration.

A configuration c of nonterminal B is **balanced** if $|c| = \varphi(B)$. This means that the number of contact points and non-contact points are the same.

The contact rank d defined in the previous section is the maximum size of any configuration of any nonterminal in any rule. For a given nonterminal B, if $\varphi(B) < d$, then we can copy entries between equivalent cells. To see this, suppose that we are moving from cell (i, j) to (k, ℓ) where the length of i is greater than the length of j. As long as we move the first index from row to column, rather than from column to row, the intermediate results will require addresses no longer than the length of i.

However, if $\varphi(B) = d$, then every configuration in which B appears is balanced:

$$\forall c \in \text{config}(B) \quad |c| = \varphi(B)$$

If $\varphi(B) = d$ and B appears in more than one configuration, that is, $|\text{config}(B)| > 1$, it is impossible to copy entries for B between the cells using a matrix of size $(2n)^d$. This is because we cannot move indices from row to column or from column to row without creating an intermediate row or column address of length greater than d as a result of the first ⬩ or ⬧ operation.

We define a **balanced grammar** to be a grammar containing a nonterminal B such that $\varphi(B) = d$, and $|\text{config}(B)| > 1$. As examples, a CFG is not balanced because, although for each nonterminal B, $\varphi(B) = d = 1$, the number of configurations $|\text{config}(B)|$ is 1. Similarly, TAG is not balanced, because each nonterminal has only one configuration. ITGs are balanced, because, for each nonterminal B, $\varphi(B) = d = 2$, and nonterminals can be used in two configurations, corresponding to straight and inverted rules.

The following condition will determine which of two alternative methods we use for the top level of our parsing algorithm.

Condition 4.1
Unbalanced Grammar Condition There is no nonterminal B such that $\varphi(B) = d$ and $|\text{config}(B)| > 1$.

This condition guarantees that we can move nonterminals as necessary with matrix multiplication:

Lemma 4
Let (i, j) and (k, ℓ) be matrix addresses such that $m(i, j) = m(k, \ell)$. Under Condition 4.1, for any nonterminal A with $|\text{config}(A)| > 1$ that appears in cell (i, j) in $T^{(n)}$, A will also appear in cell (k, ℓ) of the power matrix $T^{(n+6d)}$.

Proof
The number of A's endpoints is $2\varphi(A) = |i| + |j| = |k| + |\ell|$. If the grammar is not balanced, then $d > \varphi(A)$, and therefore $d > \min\{|i|, |j|\}$ and $d > \min\{|k|, |\ell|\}$. By Lemma 3, A will appear in cell (k, ℓ) of the power matrix $T^{(n+6d)}$. ∎

4.5 Computing the Transitive Closure of T

The transitive closure T^+ of a matrix T is the result of repeated applications of the matrix multiplication operator described in Equation (5). With T being the seed matrix, we define

$$T^+ = T^{(1)} \cup T^{(2)} \cup \cdots$$

where $T^{(i)}$ is defined recursively as:

$$T^{(1)} = T$$

$$T^{(i)} = \bigcup_{j=1}^{i-1} \left(T^{(j)} \otimes T^{(i-j)} \right)$$

Under Condition 4.1, one can show that given an LCFRS derivation tree t over the input string, each node in t must appear in the transitive closure matrix T^+. Specifically, for each node in t representing nonterminal A spanning endpoints $\{(\ell_1, \ell_2), (\ell_3, \ell_4), \ldots, (\ell_{2\varphi(A)-1}, \ell_{2\varphi(A)})\}$, at each cell $T_{i,j}^+$ in the matrix such that $m(i,j) = \{(\ell_1, \ell_2), (\ell_3, \ell_4), \ldots, (\ell_{2\varphi(A)-1}, \ell_{2\varphi(A)})\}$, contains A. This leads to the following result:

Lemma 5
Under Condition 4.1, the transitive closure of T is such that $[T^+]_{ij}$ represents the set of nonterminals that are derivable for the given spans in $m(i,j)$.

Proof
The proof is by induction over the length of the LCFRS derivations. By Lemma 1, derivations consisting of a single rule $A[\alpha] \to B[\beta] C[\gamma]$ produce $A \in T^{(2)}$ for i and j corresponding the non-combination points of B and C. For all other i and j such that $m(i,j) = \{(\ell_1, \ell_2), (\ell_3, \ell_4), \ldots, (\ell_{2\varphi(A)-1}, \ell_{2\varphi(A)})\}$, an entry is produced in $T_{ij}^{(6d+2)}$ by Lemma 4. By induction, $T^{s(6d+2)}$ contains entries for all LCFRS derivations of depth s, and T^+ contains entries for all LCFRS derivations of any length.

In the other direction, we need to show that all entries A in T^+ correspond to a valid LCFRS derivation of nonterminal A spanning endpoints $m(i,j)$. This can be shown by induction over the number of matrix multiplications. During each multiplication, entries created in the product matrix correspond either to the application of an LCFRS rule with l.h.s. A, or to the movement of an index between row and column address for a previously recognized instance of A. ∎

The transitive closure still yields a useful result, even when Condition 4.1 does not hold. To show how it is useful, we need to define the "copying" operator, Π,

which takes a matrix T' of the same type of T, and sets $\Pi(T')$ using the following procedure:

1. Define $e(i,j) = \{(i',j') \mid m(i',j') = m(i,j)\}$—that is, the set of equivalent configurations to (i,j).

2. Set $[\Pi(T')]_{ij} = \bigcup_{(i',j') \in e(i,j)} A_{i'j'}$.

This means that Π takes a completion step, and copies all nonterminals between all equivalent addresses in T'. Note that the Π operator can be implemented such that it operates in time $O(n^{2d})$. All it requires is just taking $O(n^{2d})$ unions of sets (corresponding to the sets of nonterminals in the matrix cells), where each set is of size $O(1)$ with respect to the sentence length (i.e., the size is only a function of the grammar), and each union is over $O(1)$ sets.

This procedure leads to a recognition algorithm for binary LCFRS that do not satisfy Condition 4.1 (we also assume that these binary LCFRS would not have unary cycles or ϵ rules). This algorithm is given later, in Figure 9. It operates by iterating through transitive closure steps and copying steps until convergence. When we take the transitive closure of T, we are essentially computing a subset of the derivable nonterminals. Then, the copying step (with Π) propagates nonterminals through equivalent cells. Now, if we take the transitive closure again, and there is any way to derive new nonterminals because of the copying step, the resulting matrix will have at least one new nonterminal. Otherwise, it will not change, and as such, we recognized all possible derivable nonterminals in each cell.

Lemma 6
For any single-initial LCFRS, when Step 2 of the algorithm in Figure 9 converges, T is such that $[T]_{ij}$ represents the set of nonterminals that are derivable for the given spans in $m(i,j)$.

Proof
Any LCFRS derivation of a nonterminal can be decomposed into a sequence of rule applications and copy operations, and by induction over the length of the derivation, all derivations will be found. Each matrix operation only produces derivable LCFRS nonterminals, and by induction over the number of steps of the algorithm, only derivable nonterminals will be found. ∎

4.5.1 Reduction of Transitive Closure to Boolean Matrix Multiplication. Valiant (1975) showed that his algorithm for computing the multiplication of two matrices, in terms of a multiplication operator similar to ours, can be reduced to the problem of Boolean matrix multiplication. His transitive closure algorithm requires as a black box this two-matrix multiplication algorithm.

We follow here a similar argument. We can use Valiant's algorithm for the computation of the transitive closure, because our multiplication operator is distributive (with respect to $\cup$). To complete our argument, we need to show, similarly to Valiant, that the product of two matrices using our multiplication operator can be reduced to Boolean matrix multiplication (Figure 7).

Consider the problem of multiplication a matrix T_1 and T_2, and say $T_1 \otimes T_2 = T_3$. To reduce it to Boolean matrix multiplication, we create $2|\mathcal{R}|$ pairs of matrices, G_r and H_r,

G_B (2, 7) H_C (4, 5)

(1, 8) $\begin{pmatrix} & & \\ & 1 & \\ & & \\ & & \\ & & \end{pmatrix}$ (2, 7) $\begin{pmatrix} & & \\ & 1 & \\ & & \\ & & \\ & & \end{pmatrix}$

$I_{BC} = G_B H_C$ (4, 5)

(1, 8) $\begin{pmatrix} & & \\ & 1 & \\ & & \\ & & \\ & & \end{pmatrix}$

Figure 7
Reduction of transitive closure to Boolean matrix multiplication. Boolean matrix operations
implementing the matrix multiplication example of Section 3.

where r ranges over $\mathcal{R}$. The size of G_r and H_r is $N \times N$. If $r = A[\alpha] \rightarrow B[\beta]C[\gamma]$, we set
$[G_r]_{ik}$ to be 1 if the nonterminal B appears in $[T_1]_{ik}$ and B, i, and k meet the conditions of
Step 2c of Figure 5. Similarly, we set $[H_r]_{kj}$ to be 1 if the nonterminal C appears in $[T_2]_{kj}$
and C, k, and j meet the conditions of Step 2d. All other cells, in both G_r and H_r, are set
to 0. Note that G_r and H_r for all $r \in \mathcal{R}$ are upper triangular Boolean matrices.

In addition, we create $2|\mathcal{L}|$ pairs of matrices, G_A and H_A, where A ranges over the set
of nonterminals $\mathcal{L}$. We set $[G_A]_{ik}$ to be 1 if the nonterminal A appears in $[T_1]_{ik}$, regardless
the conditions of Step 2c of Figure 5. Similarly, we set $[H_A]_{kj}$ to be 1 if the nonterminal A
appears in $[T_2]_{kj}$, regardless of the conditions of Step 2d. All other cells, in both G_A and
H_A, are set to 0. Again, G_A and H_A for all $A \in \mathcal{L}$ are upper triangular Boolean matrices.

Finally, we create six additional matrices, for each element in the set $\{$⬓,⬒,
◨,⬒,◪,▭$\}$. These matrices indicate the positions in which each symbol appears in the
seed matrix T defined in Figure 4:

1. $G_⬒$, for which $[G_⬒]_{ij} = 1$ only if $(⬒, i, j) \in T_{ij}$.

2. $H_{◨}$, for which $[H_{◨}]_{ij} = 1$ only if $(◨, i, j) \in T_{ij}$.

3. $H_{◪}$, for which $[H_{◪}]_{ij} = 1$ only if $(◪, i, j) \in T_{ij}$.

4. $G_{⬓}$, for which $[G_{⬓}]_{ij} = 1$ only if $(⬓, i, j) \in T_{ij}$.

5. $H_{◨}$, for which $[H_{◨}]_{ij} = 1$ only if $(◨, i, j) \in T_{ij}$.

6. $G_▭$, for which $[G_▭]_{ij} = 1$ only if $(▭, i, j) \in T_{ij}$.

Now, for each rule $r \in \mathcal{R}$, we compute the matrix $I_r = G_r H_r$. The total number of matrix multiplications required is $|\mathcal{R}|$, which is constant in n. Now, T_3 can be obtained by multiplying these matrices, and applying the conditions of Figure 5:

1. For each $A \in \mathcal{L}$, for each rule $r = A \to B\,C$, check whether $[I_r]_{ij} = 1$. If Step 2e is satisfied for A, i, and j, then add (A, i, j) to $[T_3]_{ij}$.

2. For each $A \in \mathcal{L}$, compute $J_A = G_A H_{\square\textbf{.}}$. For each (i, j), add A to $[T_3]_{ij}$ if $\hat{x} \in j$ and $x \in i$ for some x, and $[J_A]_{ij} = 1$.

3. For each $A \in \mathcal{L}$, compute $J_A = G_{\underline{\textbf{+}}} H_A$. For each (i, j), add A to $[T_3]_{ij}$ $\hat{x} \in j$ and $x \notin i$ and for some x, and $[J_A]_{ij} = 1$.

4. For each $A \in \mathcal{L}$, compute $J_A = G_A H_{\square}$. For each (i, j), add A to $[T_3]_{ij}$ if $|i \cup j| = 2\varphi(A)$, and $[J_A]_{ij} = 1$.

5. For each $A \in \mathcal{L}$, compute $J_A = G_{\underline{\textbf{+}}} H_A$. For each (i, j), add A to $[T_3]_{ij}$ if $\hat{x} \in i$ and $x \in j$ for some x, and $[J_A]_{ij} = 1$.

6. For each $A \in \mathcal{L}$, compute $J_A = G_A H_{\square\textbf{.}}$. For each (i, j), add A to $[T_3]_{ij}$ $\hat{x} \in i$ and $x \notin j$ for some x, and $[J_A]_{ij} = 1$.

7. For each $A \in \mathcal{L}$, compute $J_A = G_{\square} H_A$. For each (i, j), add A to $[T_3]_{ij}$ if $|i \cup j| = 2\varphi(A)$, and $[J_A]_{ij} = 1$.

Lemma 7
The matrix product operation for two matrices of size $(2n)^d \times (2n)^d$ can be computed in time $O(n^{\omega d})$, if two $m \times m$ Boolean matrices can be multiplied in time $O(m^\omega)$.

Proof
The result of the algorithm above is guaranteed to be the same as the result of matrix multiplication using the $\otimes$ operation of Figure 5 because it considers all combinations of i, j, and k and all pairs of nonterminals and copy symbols, and applies the same set of conditions. This is possible because each of the conditions in Figure 5 applies either to a pair (i, k) or (k, j), in which case we apply the condition to input matrices to the Boolean matrix multiplication, or to the pair (i, j), in which case we apply the condition to the result of the Boolean matrix multiplication. Crucially, no condition in Figure 5 involves i, j, and k simultaneously.

The Boolean matrix algorithm takes time $O(n^{\omega d})$ for each matrix multiplication, while the pre- and post-processing steps for each matrix multiplication take only $O(n^{2d})$. The number of Boolean matrix multiplications depends on the grammar, but is constant with respect to n, yielding an overall runtime of $O(n^{\omega d})$. ∎

The final parsing algorithm is given in Figure 8. It works by computing the seed matrix T, and then finding its transitive closure. Finally, it checks whether the start symbol appears in a cell with an address that spans the whole string. If so, the string is in the language of the grammar.

5. Computational Complexity Analysis

As mentioned in the previous section, the algorithm in Figure 8 finds the transitive closure of a matrix under our definition of matrix multiplication. The operations $\cup$ and $\otimes$ used in our matrix multiplication distribute. The $\otimes$ operator takes the cross
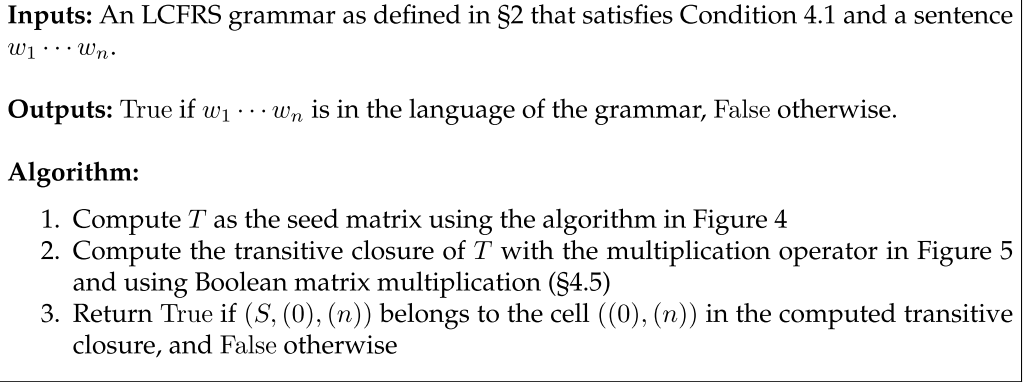

Inputs: An LCFRS grammar as defined in §2 that satisfies Condition 4.1 and a sentence $w_1 \cdots w_n$.

Outputs: True if $w_1 \cdots w_n$ is in the language of the grammar, False otherwise.

Algorithm:

1. Compute T as the seed matrix using the algorithm in Figure 4
2. Compute the transitive closure of T with the multiplication operator in Figure 5 and using Boolean matrix multiplication (§4.5)
3. Return True if $(S, (0), (n))$ belongs to the cell $((0), (n))$ in the computed transitive closure, and False otherwise

---

Figure 8
Algorithm for recognizing binary linear context-free rewriting systems when Condition 4.1 is satisfied by the LCFRS.

product of two sets, and applies a filtering condition to the results; the fact that $(x \otimes y) \cup (x \otimes z) = x \otimes (y \cup x)$ follows from the fact that it does not matter whether we take the cross product of the union, or the union of the cross product. However, unlike in the case of standard matrix multiplication, our $\otimes$ operation is not associative. In general, $x \otimes (y \otimes z) \neq (x \otimes y) \otimes z$, because the combination of y and z may be allowed by the LCFRS grammar, whereas the combination of x and y is not.

Lemma 8
The transitive closure of a matrix of size $(2n)^d \times (2n)^d$ can be computed in time $O(n^{\omega d})$, if $2 < \omega < 3$, and two $m \times m$ Boolean matrices can be multiplied in time $O(m^\omega)$.

Proof
We can use the algorithm of Valiant for finding the closure of upper triangular matrices under distributive, non-associative matrix multiplication. Because we can perform one matrix product in time $O(n^{\omega d})$ by Lemma 7, the algorithm of Valiant (1975, Theorem 2) can be used to compute transitive closure also in time $O(n^{\omega d})$. ■

When Valiant's paper was published, the best well-known algorithm known for such multiplication was Strassen's algorithm, with $M(n) = O(n^{2.8704})$. Since then, it has been found that $M(n) = O(n^\omega)$ for $\omega < 2.38$ (see also Section 1). There are ongoing attempts to further reduce ω, or find lower bounds for $M(n)$.

The algorithm for transitive closure gives one of the main results of this article:

Theorem 1
A single-initial binary LCFRS meeting Condition 4.1 can be parsed in time $O(n^{\omega d})$, where d is the contact rank of the grammar, $2 < \omega < 3$, and two $m \times m$ Boolean matrices can be multiplied in time $O(m^\omega)$.

Proof
By Lemma 8, Step 2 of the algorithm in Figure 8 takes $O(n^{\omega d})$. By Lemma 5, the result of Step 2 gives all nonterminals that are derivable for the given spans in $m(i, j)$. ■

Parsing a binary LCFRS rule with standard chart parsing techniques requires time $O(n^{\varphi(A)+\varphi(B)+\varphi(C)})$. Let $p = \max_{A \to BC \in \mathcal{R}} (\varphi(A) + \varphi(B) + \varphi(C))$. The worst-case complexity of LCFRS chart parsing techniques is $O(n^p)$. We can now ask the question: In which case the algorithm in Figure 8 is asymptotically more efficient than standard chart parsing techniques with respect to n? That is, in which cases is $n^{d\omega} = o(n^p)$?

Clearly, this would hold whenever $d\omega < p$. By definition of d and p, a sufficient condition for that is that for any rule $A \to BC \in \mathcal{R}$ it holds that:[4]

$$\max \left\{ \begin{array}{l} \varphi(A) + \varphi(B) - \varphi(C), \\ \varphi(A) - \varphi(B) + \varphi(C), \\ -\varphi(A) + \varphi(B) + \varphi(C) \end{array} \right\} < \frac{1}{\omega} (\varphi(A) + \varphi(B) + \varphi(C))$$

This means that for any rule, the following conditions should hold:

$$\omega(\varphi(A) + \varphi(B) - \varphi(C)) < \varphi(A) + \varphi(B) + \varphi(C)$$

$$\omega(\varphi(A) - \varphi(B) + \varphi(C)) < \varphi(A) + \varphi(B) + \varphi(C)$$

$$\omega(-\varphi(A) + \varphi(B) + \varphi(C)) < \varphi(A) + \varphi(B) + \varphi(C)$$

Algebraic manipulation shows that this is equivalent to having:

$$\varphi(A) + \varphi(B) < \left(\frac{\omega + 1}{\omega - 1} \right) \varphi(C)$$

$$\varphi(B) + \varphi(C) < \left(\frac{\omega + 1}{\omega - 1} \right) \varphi(A)$$

$$\varphi(C) + \varphi(A) < \left(\frac{\omega + 1}{\omega - 1} \right) \varphi(B)$$

For the best well-known algorithm for matrix multiplication, it holds that:

$$\frac{\omega + 1}{\omega - 1} > 2.44$$

For Strassen's algorithm, it holds that:

$$\frac{\omega + 1}{\omega - 1} > 2.06$$

We turn now to analyze the complexity of the algorithm in Figure 9, giving the main result of this article for arbitrary LCFRS:

Theorem 2
A single-initial binary LCFRS can be parsed in time $O(n^{\omega d+1})$, where d is the contact rank of the grammar, $2 < \omega < 3$, and two $m \times m$ Boolean matrices can be multiplied in time $O(m^\omega)$.

---

4 For two sets of real numbers, X and Y, it holds that if for all $x \in X$ there is a $y \in Y$ such that $x < y$, then $\max X < \max Y$.

Inputs: An LCFRS grammar as defined in §2 and a sentence $w_1 \cdots w_n$.

Outputs: True if $w_1 \cdots w_n$ is in the language of the grammar, False otherwise.

Algorithm:
1. Compute T as the seed matrix using the algorithm in Figure 4
2. Repeat until T does not change: $T \leftarrow (\Pi(T))^+$
3. Return True if $(S, (0), (n))$ belongs to the cell $((0), (n))$ in the computed transitive closure, and False otherwise

---

Figure 9
Algorithm for recognizing binary LCFRS when Condition 4.1 is not necessarily satisfied by the LCFRS.

Proof

The algorithm of Figure 9 works by iteratively applying the transitive closure and the copying operator until convergence. At convergence, we have recognized all derivable nonterminals by Lemma 6. Each transitive closure has the asymptotic complexity of $O(n^{\omega d})$ by Lemma 8. Each Π application has the asymptotic complexity of $O(n^{2d})$. As such, the total complexity is $O(tn^{\omega d})$, where t is the number of iterations required to converge. At each iteration, we discover at least one new nonterminal. The total number of nodes in the derivation for the recognized string is $O(n)$ (assuming no unary cycles or ϵ rules). As such $t = O(n)$, and the total complexity of this algorithm is $O(n^{\omega d+1})$. ∎

6. Applications

Our algorithm is a recognition algorithm that is applicable to binary LCFRS. As such, our algorithm can be applied to any LCFRS, by first reducing it to a binary LCFRS. We discuss results for specific classes of LCFRS in this section, and return to the general binarization process in Section 7.6.

LCFRS subsumes context-free grammars, which was the formalism that Valiant (1975) focused on. Valiant showed that the problem of CFG recognition can be reduced to the problem of matrix multiplication, and, as such, the complexity of CFG recognition in that case is $O(n^{\omega})$. Our result generalizes Valiant's result. CFGs (in Chomsky Normal Form) can be reduced to a binary LCFRS with $f = 1$. As such, $d = 1$ for CFGs, and our algorithm yields a complexity of $O(n^{\omega})$. (Note that CFGs satisfy Condition 4.1, and therefore we can use a single transitive closure step.)

LCFRS is a broad family of grammars, and it subsumes many other well-known grammar formalisms, some of which were discovered or developed independently of LCFRS. Two such formalisms are tree-adjoining grammars (Joshi and Schabes 1997) and synchronous context-free grammars. In the next two sections, we explain how our algorithmic result applies to these two formalisms.

6.1 Mildly Context-Sensitive Language Recognition

Linear context-free rewriting systems fall under the realm of mildly context-sensitive grammar formalisms. They subsume four important mildly context-sensitive

formalisms that were developed independently and later shown to be weakly equivalent by Vijay-Shanker and Weir (1994): tree-adjoining grammars (Joshi and Schabes 1997), linear indexed grammars (Gazdar 1988), head grammars (Pollard 1984), and combinatory categorial grammars (Steedman 2000). Weak equivalence here refers to the idea that any language generated by a grammar in one of these formalisms can be also generated by some grammar in any of the other formalisms among the four. It can be verified that all of these formalisms are unbalanced, single-initial LCFRSs, and as such, the algorithm in Figure 8 applies to them.

Rajasekaran and Yooseph (1998) show that tree-adjoining grammars can be parsed with an asymptotic complexity of $O(M(n^2)) = O(n^{4.76})$. Although they did not discuss that, the weak equivalence between the four formalisms mentioned here implies that all of them can be parsed in time $O(M(n^2))$. Our algorithm generalizes this result. We now give the details.

Our starting point for this discussion is head grammars. Head grammars are a specific case of linear context-free rewriting systems, not just in the formal languages they define—but also in the way these grammars are described. They are described using concatenation production rules and wrapping production rules, which are directly transferable to LCFRS notation. Their fan-out is 2. We focus in this discussion on "binary head grammars," defined analogously to binary LCFRS—the rank of all production rules has to be 2. The contact rank of binary head grammars is 2. As such, our work shows that the complexity of recognizing binary head grammar languages is $O(M(n^2)) = O(n^{4.76})$.

Vijay-Shanker and Weir (1994) show that linear indexed grammars (LIGs) can actually be reduced to binary head grammars. Linear indexed grammars are extensions of CFGs, a linguistically motivated restricted version of indexed grammars, the latter of which were developed by Aho (1968) for the goal of handling variable binding in programming languages. The main difference between LIGs and CFGs is that the nonterminals carry a "stack," with a separate set of stack symbols. Production rules with LIGs copy the stack on the left-hand side to *one* of the nonterminal stacks in the right-hand side,[5] potentially pushing or popping one symbol in the new copy of the stack. For our discussion, the main important detail about the reduction of LIGs to head grammars is that it preserves the rank of the production rules. As such, our work shows that binary LIGs can also be recognized in time $O(n^{4.76})$.

Vijay-Shanker and Weir (1994) additionally address the issue of reducing combinatory categorial grammars to LIGs. The combinators they allow are function application and function composition. The key detail here is that their reduction of CCG is to an LIG with rank 2, and, as such, our algorithm applies to CCGs as well, which can be recognized in time $O(n^{4.76})$.

Finally, Vijay-Shanker and Weir (1994) reduced tree-adjoining grammars to combinatory categorial grammars. The TAGs they tackle are in "normal form," such that the auxiliary trees are binary (all TAGs can be reduced to normal form TAGs). Such TAGs can be converted to weakly equivalent CCG (but not necessarily strongly equivalent), and as such, our algorithm applies to TAGs as well. As mentioned earlier, this finding supports the finding of Rajasekaran and Yooseph (1998), who show that TAG can be recognized in time $O(M(n^2))$.

For an earlier discussion connections between TAG parsing and Boolean matrix multiplication, see Satta (1994).

---

5 General indexed grammars copy the stack to multiple nonterminals on the right-hand side.

6.2 Synchronous Context-Free Grammars

SCFGs are widely used in machine translation to model the simultaneous derivation of translationally equivalent strings in two natural languages, and are equivalent to the syntax-directed translation schemata of Aho and Ullman (1969). SCFGs are a subclass of LCFRS where each nonterminal has fan-out 2: one span in one language and one span in the other. Because the first span of the l.h.s. nonterminal always contains spans from both r.h.s. nonterminals, SCFGs are always single-initial. Binary SCFGs, also known as ITGs, have no more than two nonterminals on the r.h.s. of a rule, and are the most widely used model in syntax-based statistical machine translation.

Synchronous parsing with traditional tabular methods for ITG is $O(n^6)$, as each of the three nonterminals in a rule has fan-out of two. ITGs, unfortunately, do not satisfy Condition 4.1, and therefore we have to use the algorithm in Figure 9. Still, just like with TAG, each rule combines two nonterminals of fan-out 2 using two combination points. Thus, $d = 2$, and we achieve a bound of $O(n^{2\omega+1})$ for ITG, which is $O(n^{5.76})$ using the current state of the art for matrix multiplication.

We achieve even greater gains for the case of multi-language synchronous parsing. Generalizing ITG to allow two nonterminals on the right-hand side of a rule in each of k languages, we have an LCFRS with fan-out k. Traditional tabular parsing has an asymptotic complexity of $O(n^{3k})$, whereas our algorithm has the complexity of $O(n^{\omega k+1})$.

Another interesting case of a synchronous formalism that our algorithm improves the best well-known result for is that of binary synchronous TAGs (Shieber and Schabes 1990)—that is, a TAG in which all auxiliary trees are binary. This formalism can be reduced to a binary LCFRS. A tabular algorithm for such grammar has the asymptotic complexity of $O(n^{12})$. With our algorithm, $d = 4$ for this formalism, and as such its asymptotic complexity in that case is $O(n^{9.52})$.

7. Discussion and Open Problems

In this section, we discuss some extensions to our algorithm and open problems.

7.1 Turning Recognition into Parsing

The algorithm we presented focuses on recognition: Given a string and a grammar, it can decide whether the string is in the language of the grammar or not. From an application perspective, perhaps a more interesting algorithm is one that returns an actual derivation tree, if it identifies that the string is in the language.

It is not difficult to adapt our algorithm to return such a parse, without changing the asymptotic complexity of $O(n^{\omega d+1})$. Once the transitive closure of T is computed, we can backtrack to find such a parse, starting with the start symbol in a cell spanning the whole string. When we are in a specific cell, we check all possible combination points (there are d of those) and nonterminals, and if we find such pairs of combination points and nonterminals that are valid in the chart, then we backtrack to the corresponding cells. The asymptotic complexity of this post-processing step is $O(n^{d+1})$, which is less than $O(n^{\omega d})$ ($\omega > 2, d > 1$).

This post-processing step corresponds to an algorithm that finds a parse tree, *given* a pre-calculated chart. If the chart was not already available when our algorithm finishes, the asymptotic complexity of this step would correspond to the asymptotic complexity of a naïve tabular parsing algorithm. It remains an open problem to adapt our algorithm to *probabilistic parsing*, for example—finding the highest scoring parse given a
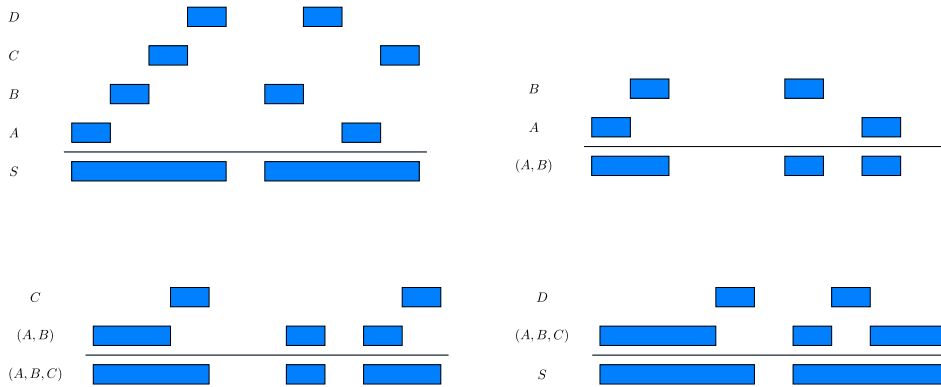
Figure 10
Upper left: Combination of spans for SCFG rule $[S \rightarrow ABCD,\ BDAC]$. Upper right and bottom row: Three steps in parsing binarized rule.

probabilistic or a weighted LCFRS (Kallmeyer and Maier 2010). See more details in Section 7.3.

7.2 General Recognition for Synchronous Parsing

Similarly to LCFRS, the rank of an SCFG is the maximal number of nonterminals that appear in the right-hand side of a rule. Any SCFG can be binarized into an LCFRS grammar. However, when the SCFG rank is arbitrary, this means that the fan-out of the LCFRS grammar can be larger than 2. This happens because binarization creates intermediate nonterminals that span several substrings, denoting binarization steps of the rule. These substrings are eventually combined into two spans, to yield the language of the SCFG grammar (Huang et al. 2009).

Our algorithm does not always improve the asymptotic complexity of SCFG parsing over tabular methods. For example, Figure 10 shows the combination of spans for the rule $[S \rightarrow ABCD, BDAC]$, along with a binarization into three simpler LCFRS rules. A naïve tabular algorithm for this rule would have the asymptotic complexity of $O(n^{10})$, but the binarization shown in Figure 10 reduces this to $O(n^8)$. Our algorithm gives a complexity of $O(n^{9.52})$, as the second step in the binarization shown consists of a rule with $d = 4$.

7.3 Generalization to Weighted Logic Programs

Weighted logic programs (WLPs) are declarative programs, in the form of Horn clauses similar to those that Prolog uses, that can be used to formulate parsing algorithms such as CKY and other types of dynamic programming algorithms or NLP inference algorithms (Eisner, Goldlust, and Smith 2005; Cohen, Simmons, and Smith 2011).

For a given Horn clause, WLPs also require a "join" operation that sums (in some semiring) over a set of possible values in the free variables in the Horn clauses. With CKY, for example, this sum will be performed on the mid-point concatenating two spans. This join operation is also the type of operation we address in this paper (for LCFRS) in order to improve their asymptotic complexity.

It remains an open question to see whether we can generalize our algorithm to arbitrary weighted logic programs. In order to create an algorithm that takes as input

a weighted logic program (and a set of axioms) and "recognizes" whether the goal is achievable, we would need to have a generic way of specifying the set N, which was specialized to LCFRS in this case. Not only that, we would have to specify N in such a way that the asymptotic complexity of the WLP would improve over a simple dynamic programming algorithm (or a memoization technique).

In addition, in this paper we focus on the problem of recognition and parsing for unweighted grammars. Benedí and Sánchez (2007) showed how to generalize Valiant's algorithm in order to compute inside probabilities for a PCFG and a string. Even if we were able to generalize our addressing scheme to WLPs, it remains an open question to see whether we can go beyond recognition (or unweighted parsing).

7.4 Rytter's Algorithm

Rytter (1995) gives an algorithm for CFG parsing with the same time complexity as Valiant's, but a somewhat simpler divide-and-conquer strategy. Rytter's algorithm works by first recursively finding all chart items entirely within the first half of the string and entirely within the second half of the string. The combination step uses a shortest path computation to identify the sequence of chart items along a spine of the final parse tree, where the spine extends from the root of the tree to the terminal in position $n/2$. Rytter's algorithm relies on the fact that this spine, consisting of chart items that cross the midpoint of the string, forms a single path from the root to one leaf of the derivation tree. This property does not hold for general LCFRS, because two siblings in the derivation tree may both correspond to multiple spans in the string, each containing material on both sides of the string midpoint. For this reason, Rytter's algorithm does not appear to generalize easily to LCFRS.

7.5 Relation to Multiple Context-Free Grammars

Nakanishi et al. (1998) develop a matrix multiplication parsing algorithm for multiple context-free grammars (MCFGs). When these grammars are given in a binary form, they can be reduced to binary LCFRS. Similarly, binary LCFRS can be reduced to binary MCFGs. The algorithm that Nakanishi et al. develop is simpler than ours, and does not directly tackle the problem of transitive closure for LCFRS. More specifically, Nakanishi et al. multiply a seed matrix such as our T by itself in several steps, and then follow up with a copying operation between equivalent cells. They repeat this n times, where n is the sentence length. As such, the asymptotic complexity of their algorithm is identical for both balanced and unbalanced grammars, a distinction they do not make.

The complexity analysis of Nakanishi et al. is different from ours, but in certain cases, yields identical results. For example, if $\varphi(a) = f$ for all $a \in \mathcal{L}$, and the grammar is balanced, then both our algorithm and their algorithm give a complexity of $O(n^{\omega f + 1})$. If the grammar is unbalanced, then our algorithm gives a complexity of $O(n^{\omega f})$, whereas the asymptotic complexity of their algorithm remains $O(n^{\omega f + 1})$. As such, Nakanishi et al.'s algorithm does not generalize Valiant's algorithm—its asymptotic complexity for context-free grammars is $O(n^{\omega + 1})$ and not $O(n^{\omega})$.

Nakanishi et al. pose in their paper an open problem, which loosely can be re-worded as the problem of finding an algorithm that computes the transitive closure of T without the extra $O(n)$ factor that their algorithm incurs. In our paper, we provide a solution to this open problem for the case of single-initial, unbalanced grammars. The core of the solution lies in the matrix multiplication copying mechanism described in Section 4.2.

7.6 Optimal Binarization Strategies

The two main grammar parameters that affect the asymptotic complexity of parsing with LCFRS (in its general form) are the fan-out of the nonterminals and the rank of the rules. With tabular parsing, we can actually refer to the parsing complexity of a *specific rule in the grammar*. Its complexity is $O(n^p)$, where the parsing complexity p is the total fan-out of all nonterminals in the rule. For binary rules of the form $A \to BC$, $p = \varphi(A) + \varphi(B) + \varphi(C)$.

To optimize the tabular algorithm time complexity of parsing with a binary LCFRS, equivalent to another non-binary LCFRS, we would want to minimize the time complexity it takes to parse each rule. As such, our goal is to minimize $\varphi(A) + \varphi(B) + \varphi(C)$ in the resulting binary grammar. Gildea (2011) has shown that this metric corresponds to the tree width of a dependency graph that is constructed from the grammar. It is not known whether finding the optimal binarization of an LCFRS is an NP-complete problem, but Gildea shows that a polynomial time algorithm would imply improved approximation algorithms for the treewidth of general graphs.

In general, the optimal binarization for tabular parsing may not by the same as the optimal binarization for parsing with our algorithm based on matrix multiplication. In order to optimize the complexity of our algorithm, we want to minimize d, which is the maximum over all rules $A \to BC$ of

$$d(A \to BC) = \max\{\varphi(A) + \varphi(B) - \varphi(C), \varphi(A) - \varphi(B) + \varphi(C), -\varphi(A) + \varphi(B) + \varphi(C)\}$$

For a fixed binarized grammar, d is always less than p, the tabular parsing complexity, and, hence, the optimal d^* over binarizations of an LCFRS is always less than the optimal p^* for tabular parsing. However, whether any savings can be achieved with our algorithm depends on whether $\omega d^* < p^*$, or $\omega d^* + 1 < p^*$ in the case of balanced grammars. Our criterion does not seem to correspond closely to a well-studied graph-theoretic concept such a treewidth, and it remains an open problem to find an efficient algorithm that minimizes this definition of parsing complexity.

It is worth noting that $d(A \to BC) \geq \frac{1}{3}(\varphi(A) + \varphi(B) + \varphi(C))$. As such, this gives a lower bound on the time complexity of our algorithm relative to tabular parsing using the same binarized grammar. If $O(n^{t_1})$ is the asymptotic complexity of our algorithm, and $O(n^{t_2})$ is the asymptotic complexity of a tabular algorithm, then $\frac{t_1}{t_2} \geq \frac{\omega}{3} > 0.79$.

8. Conclusion

We described a parsing algorithm for binary linear context-free rewriting systems that has the asymptotic complexity of $O(n^{\omega d+1})$ where $\omega < 2.38$, d is the "contact rank" of the grammar (the maximal number of combination points in the rules in the grammar in single-initial form), and n is the string length. Our algorithm has the asymptotic complexity of $O(n^{\omega d})$ for a subset of binary LCFRS that are unbalanced. Our result generalizes the algorithm of Valiant (1975), and also reinforces existing results about mildly context-sensitive parsing for tree-adjoining grammars (Rajasekaran and Yooseph 1998). Our result also implies that inversion transduction grammars can be parsed in time $O(n^{2\omega+1})$ and that synchronous parsing with k languages has the asymptotic complexity of $O(n^{\omega k+1})$ where k is the number of languages.

Appendix A. Notation

Table A.1 provides a table of notation for symbols used in this article.

Table A.1
Table of notation symbols used in this article.

Symbol	Description	1st mention
$M(n)$	The complexity of Boolean $n \times n$ matrix multiplication	§1
ω	Best well-known complexity for $M(n)$, $M(n) = O(n^{\omega})$	§1
$[n]$	Set of integers $\{1, \dots, n\}$	§2
$[n]_0$	$[n] \cup \{0\}$	§2
$\mathcal{L}$	Nonterminals of the LCFRS	§2
$\mathcal{T}$	Terminal symbols of the LCFRS	§2
$\mathcal{V}$	Variables that denote spans in grammar	§2
$\mathcal{R}$	Rules in the LCFRS	§2
A,B,C	Nonterminals	§2
f	Maximal fan-out of the LCFRS	Eq. (3)
$\varphi(A)$	Fan-out of nonterminal A	§2
y	Denotes a variable in $\mathcal{V}$ (potentially subscripted)	§2
T	Seed matrix	§3
$N, N(d)$	Set of indices for addresses in the matrix	Eq. (4)
i, j	Indices for cells in T. $i, j \in N$	§4.1
d	Grammar contact rank	§4.1
M	T_{ij} is a subset of M	§4.1
♙,♙,☐	Copying/marking symbols for rows	§4.1
▯▸,▯▸,▯	Copying/marking symbols for columns	§4.1
n	Length of sentence to be parsed	§1
$<$	Total order between the set of indices of T	§4.1
$m(i, j)$	Merged sorted sequence of $i \cup j$, divided into pairs	§4.1
remove(v, x)	Removal of x from a sequence v	Figure 5
insert(v, x)	Insertion of x in a sequence v	§4.5
Π	Copying operator	§4.5

References

Abboud, Amir, Arturs Backurs, and Virginia Vassilevska Williams. 2015. If the current clique algorithms are optimal, so is Valiant's parser. *arXiv preprint arXiv:1504.01431*.

Aho, Alfred V. 1968. Indexed grammars—an extension of context-free grammars. *Journal of the ACM*, 15(4):647–671.

Aho, Alfred V. and Jeffery D. Ullman. 1969. Syntax directed translations and the

pushdown assembler. *Journal of Computer and System Sciences*, 3:37–56.

Benedí, José-Miguel and Joan-Andreu Sánchez. 2007. Fast stochastic context-free parsing: A stochastic version of the Valiant algorithm. In J. Martí, A. M. Mendonça, and J. Serat, editors, *Pattern Recognition and Image Analysis*. Springer, pages 80–88.

Cocke, John and Jacob T. Schwartz. 1970. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University.

Cohen, Shay B., Giorgio Satta, and Michael Collins. 2013. Approximate PCFG parsing using tensor decomposition. In *Proceedings of the 2013 Meeting of the North American chapter of the Association for Computational*

Linguistics (NAACL-13), pages 487–496, Atlanta, GA.

Cohen, Shay B., Robert J. Simmons, and Noah A. Smith. 2011. Products of weighted logic programs. *Theory and Practice of Logic Programming*, 11(2–3):263–296.

Coppersmith, D. and S. Winograd. 1987. Matrix multiplication via arithmetic progressions. In *Proceedings of the 19th Annual ACM Conference on Theory of Computing*, pages 1–6, New York, NY.

Dunlop, Aaron, Nathan Bodenstab, and Brian Roark. 2010. Reducing the grammar constant: An analysis of CYK parsing efficiency. Technical report CSLU-2010-02, OHSU.

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

Eisner, Jason, Eric Goldlust, and Noah A. Smith. 2005. Compiling Comp Ling: Practical weighted dynamic programming and the Dyna language. In *Proceedings of HLT-EMNLP*, pages 281–290, Vancouver.

Eisner, Jason and Giorgio Satta. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 457–464, Baltimore, MD.

Gazdar, Gerald. 1988. *Applicability of indexed grammars to natural languages*. Springer.

Gildea, Daniel. 2011. Grammar factorization by tree decomposition. *Computational Linguistics*, 37(1):231–248.

Huang, Liang, Hao Zhang, Daniel Gildea, and Kevin Knight. 2009. Binarization of synchronous context-free grammars. *Computational Linguistics*, 35(4):559–595.

Joshi, Aravind K. and Yves Schabes. 1997. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal languages*. Springer, pages 69–123.

Kallmeyer, Laura. 2010. *Parsing Beyond Context-Free Grammars*. Cognitive Technologies. Springer.

Kallmeyer, Laura and Wolfgang Maier. 2010. Data-driven parsing with probabilistic linear context-free rewriting systems. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING 2010)*, pages 537–545, Beijing.

Kasami, Tadao. 1965. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Lab.

Le Gall, François. 2014. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 296–303, New York, NY.

Nakanishi, Ryuichi, Keita Takada, Hideki Nii, and Hiroyuki Seki. 1998. Efficient recognition algorithms for parallel multiple context-free languages and for multiple context-free languages. *IEICE TRANSACTIONS on Information and Systems*, 81(11):1148–1161.

Pollard, Carl J. 1984. *Generalized Phrase Structure Grammars, Head Grammars and Natural Languages*. Ph.D. thesis, Stanford University.

Rajasekaran, Sanguthevar and Shibu Yooseph. 1998. TAL parsing in $O(M(n^2))$ time. *Journal of Computer and System Sciences*, 56:83–89.

Raz, Ran. 2002. On the complexity of matrix product. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 144–151, Montreal.

Rytter, Wojciech. 1995. Context-free recognition via shortest paths computation: A version of Valiant's algorithm. *Theoretical Computer Science*, 143(2):343–352.

Satta, Giorgio. 1992. Recognition of linear context-free rewriting systems. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics*, pages 89–95, Newark, DE.

Satta, Giorgio. 1994. Tree-adjoining grammar parsing and boolean matrix multiplication. *Computational Linguistics*, 20(2):173–191.

Shieber, S. M. 1985. Evidence against the context-freeness of natural language. In *Linguistics and Philosophy*, volume 8. D. Reidel Publishing Company, pages 333–343.

Shieber, Stuart M and Yves Schabes. 1990. Synchronous tree-adjoining grammars. In *Proceedings of the 13th Conference on Computational Linguistics-Volume 3*, pages 253–258, Stroudsburg, PA.

Steedman, Mark. 2000. *The Syntactic Process*. Language, speech, and communication. MIT Press, Cambridge, MA.

Strassen, V. 1969. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356.

Valiant, Leslie G. 1975. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10:308–315.

Vijay-Shanker, K. and David Weir. 1994. The equivalence of four extensions of context-free grammars.

Mathematical Systems Theory, 27:511–546.

Wu, Dekai. 1997. Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. *Computational Linguistics*, 23(3):377–403.

Younger, Daniel H. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.