Approximating Language Edit Distance Beyond Fast Matrix Multiplication: Ultralinear Grammars Are Where Parsing Becomes Hard!

Rajesh Jayaram[1] **and Barna Saha***[2]

1 Brown University
`rajesh_jayaram@brown.edu`

2 University of Massachusetts Amherst
`barna@cs.umass.edu`

Abstract

In 1975, a breakthrough result of L. Valiant showed that parsing context free grammars can be reduced to Boolean matrix multiplication, resulting in a running time of $O(n^\omega)$ for parsing where $\omega \leq 2.373$ is the exponent of fast matrix multiplication, and n is the string length. Recently, Abboud, Backurs and V. Williams (FOCS 2015) demonstrated that this is likely optimal; moreover, a combinatorial $o(n^3)$ algorithm is unlikely to exist for the general parsing problem[1]. The language edit distance problem is a significant generalization of the parsing problem, which computes the minimum edit distance of a given string (using insertions, deletions, and substitutions) to any valid string in the language, and has received significant attention both in theory and practice since the seminal work of Aho and Peterson in 1972. Clearly, the lower bound for parsing rules out any algorithm running in $o(n^\omega)$ time that can return a nontrivial multiplicative approximation of the language edit distance problem. Furthermore, combinatorial algorithms with cubic running time or algorithms that use fast matrix multiplication are often not desirable in practice.

To break this n^ω hardness barrier, in this paper we study *additive* approximation algorithms for language edit distance. We provide two explicit combinatorial algorithms to obtain a string with minimum edit distance with performance dependencies on either the number of *non-linear productions*, k^*, or the number of *nested non-linear production*, k, used in the optimal derivation. Explicitly, we give an additive $O(k^*\gamma)$ approximation in time $O(|G|(n^2 + \frac{n^3}{\gamma^3}))$ and an additive $O(k\gamma)$ approximation in time $O(|G|(n^2 + \frac{n^3}{\gamma^2}))$, where $|G|$ is the grammar size and n is the string length. In particular, we obtain tight approximations for an important subclass of context free grammars known as *ultralinear* grammars, for which k and k^* are naturally bounded. Interestingly, we show that the same conditional lower bound for parsing context free grammars holds for the class of ultralinear grammars as well, clearly marking the boundary where parsing becomes hard!

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Approximation, Edit Distance, Dynamic Programming, Context Free Grammar, Hardness.

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

* Research supported by NSF CRII 1464310, NSF CAREER 1652303, a Google Faculty Research Award, and a Yahoo ACE Award.

[1] with any polynomial dependency on the grammar size

© Rajesh Jayaram and Barna Saha;
licensed under Creative Commons License CC-BY
42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Acces; Article No. 23; pp. 23:1–23:14
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Introduced by Chomsky in 1956 [11], context-free grammars (CFG) play a fundamental role in the development of formal language theory [2, 22], compiler optimization [16, 42], natural language processing [27, 31], with diverse applications in areas such as computational biology [39], machine learning [33, 20, 41] and databases [23, 14, 34]. Parsing CFG is a basic computer science question, that given a CFG G over an alphabet Σ, and a string $x \in \Sigma^*$, $|x| = n$, determines if x belongs to the language $\mathcal{L}(G)$ generated by G. The canonical parsing algorithms such as Cocke-Younger-Kasimi (CYK) [2], Earley parser, [12] etc. are based on a natural dynamic programming, and run in $O(n^3)$ time[2]. In 1975, in a theoretical breakthrough, Valiant gave a reduction from parsing to Boolean matrix multiplication, showing that the parsing problem can be solved in $O(n^\omega)$ time [38]. Despite decades of efforts, these running times have remain completely unchanged.

Nearly three decades after Valiant's result, Lee came up with an ingenious reduction from Boolean matrix multiplication to CFG parsing, showing for the first time why known parsing algorithms may be optimal [25]. A remarkable recent result of Abboud, Backurs and V. Williams made her claims concrete [1]. Based on a conjecture of the hardness of computing large cliques in graphs, they ruled out any improvement beyond Valiant's algorithm; moreover they showed that there can be no combinatorial algorithm for CFG parsing that runs in truly subcubic $O(n^{3-\epsilon})$ time for $\epsilon > 0$ [1]. However combinatorial algorithms with cubic running time or algorithms that use fast matrix multiplication are often impractical. Therefore, a long-line of research in the parsing community has been to discover subclasses of context free grammars that are sufficiently expressive yet admit efficient parsing time [26, 24, 17]. Unfortunately, there still exist important subclasses of the CFG's for which neither better parsing algorithms are known, nor have conditional lower bounds been proven to rule out the possibility of such algorithms.

Language Edit Distance.

A generalization of CFG parsing, introduced by Aho and Peterson in 1972 [3], is *language edit distance* (LED) which can be defined as follows.

▶ **Definition 1** (Language Edit Distance (LED)). Given a formal language $\mathcal{L}(G)$ generated by a grammar G over alphabet Σ, and a string $\overline{x} \in \Sigma^*$, compute the minimum number of edits (insertion, deletion and substitution) needed on $\overline{x}$ to convert it to a valid string in $\mathcal{L}(G)$.

LED is among the most fundamental and best studied problems related to strings and grammars [3, 30, 34, 35, 8, 1, 32, 6, 21], and generalizes two basic problems in computer science: parsing and string edit distance computation. Aho and Peterson presented a dynamic programming algorithm for LED that runs in $O(|G|^2 n^3)$ time [3], which was improved to $O(|G|n^3)$ by Myers in 1985 [30]. Only recently these bounds have been improved by Bringmann, Grandoni, Saha, and V. Williams to give the first truly subcubic $O(n^{2.8244})$ algorithm for LED [8]. When considering approximate answers, a *multiplicative* $(1 + \epsilon)$-approximation for LED has been presented by Saha in [35], that runs in $O(\frac{n^\omega}{\text{poly}(\epsilon)})$ time.

These subcubic algorithms for LED crucially use fast matrix multiplication, and hence are not practical. Due to the hardness of parsing [25, 1], LED cannot be approximated

[2] Dependency on the grammar size if not specified is either $|G|$ as in most combinatorial algorithms, or $|G|^2$ as in most algebraic algorithms. In this paper the algorithms will depend on $|P|$, the number of productions in the grammar. In general we assume $|P| \in \Theta(|G|)$.

with any multiplicative factor in time $o(n^\omega)$. Moreover, there cannot be any combinatorial multiplicative approximation algorithm that runs in $O(n^{3-\epsilon})$ time for any $\epsilon > 0$ [1]. LED provides a very generic framework for modeling problems with vast applications [23, 20, 41, 28, 33, 31, 15]. A fast exact or approximate algorithm for it is likely to have tangible impact, yet there seems to be a bottleneck in improving the running time beyond $O(n^\omega)$, or even in designing a truly subcubic combinatorial approximation algorithm. Can we break this n^ω barrier?

One possible approach is to allow for an *additive approximation*. Since the hardness of multiplicative approximation arise from the lower bound of parsing, it is possible to break the n^ω barrier by designing a purely combinatorial algorithm for LED with an additive approximation. Such a result will have immense theoretical and practical significance. Due to the close connection of LED with matrix products, all-pairs shortest paths and other graph algorithms [35, 8], this may imply new algorithms for many other fundamental problems. In this paper, we make a significant progress in this direction by providing the first nontrivial additive approximation for LED that runs in quadratic time. Let $G = (Q, \Sigma, P, S)$ denote a context free grammar, where Q is the set of nonterminals, Σ is the alphabet or set of terminals, P is the set of productions, and S is the starting non-terminal.

▶ **Definition 2.** Given $G = (Q, \Sigma, P, S)$, a production $A \to \alpha$ is said to be *linear* if there is at most one non-terminal in α where $A \in Q$ and $\alpha \in (Q \cup \Sigma)^*$. Otherwise, if α contains two or more non-terminals, then $A \to \alpha$ is said to be *non-linear*.

The performance of our algorithms depends on either the total number of *non-linear productions* or the maximum number of *nested non-linear productions* (depth of the parse tree after condensing every consecutive sequence of linear productions, see the full version for more details) in the derivation of string with optimal edit distance, where the latter is often substantially smaller. Explicitly, we give an additive $O(k^*\gamma)$ approximation in time $O(|G|(n^2 + \frac{n^3}{\gamma^3}))$ and an additive $O(k\gamma)$ approximation in time $O(|G|(n^2 + \frac{n^3}{\gamma^2}))$, where k^* is the number of non-linear productions in the derivation of the optimal string, and k is the maximum number of nested non-linear productions in the derivation of the optimal string (each minimized over all possible derivations). Our algorithms will be particularly useful for an important subclass of CFGs, known as the *ultralinear grammars*, for which these values are tightly bounded for all derivations [43, 10, 26, 7, 29].

▶ **Definition 3** (ultralinear)**.** A grammar $G = (Q, \Sigma, P, S)$ is said to be **k-ultralinear** if there is a partition $Q = Q_1 \cup Q_1 \cup \cdots \cup Q_k$ such that for every $X \in Q_i$, the productions of X consist of *linear productions* $X \to \alpha A | A\alpha | \alpha$ for $A \in Q_j$ with $j \leq i$ and $\alpha \in \Sigma$, or *non-linear productions* of the form $X \to w$, where $w \in (Q_1 \cup Q_2 \cup \cdots \cup Q_{i-1})^*$.

The parameter k places a built-in upper bound on the number of nested non-linear productions allowed in any derivation. Thus for simplicity we will use k both to refer to the parameter of an ultralinear grammar, as well as the maximum number of nested non-linear productions. Furthermore, if d is the maximum number of non-terminals on the RHS of a production, then d^k is a built-in upper bound on the total number of non-linear productions in any derivation. In all our algorithms, without loss of generality, we use a standard normal form where $d = 2$ for all non-linear productions. As we will see later, given any CFG G and any $k \geq 1$, we can create a new grammar G' by making k copies $Q_1, \ldots, Q_k$ of the set of non-terminals Q of G, and forcing every nonlinear production in Q_i to go to non-terminals in Q_{i-1}. Thus G' has non-terminal set $Q_1 \cup Q_2 \cup \cdots \cup Q_k$, and size $O(k|G|)$. In this way we can restrict any CFG to a k-ultralinear grammar which can produce any string in $\mathcal{L}(G)$

requiring no more than k nested non-linear productions. It is precisely this procedure of creating a k-ultralinear grammar from a CFG G that we use in our proof of hardness for parsing ultralinear languages (see the full version).

For example, if G is the well-known *Dyck Languages* [34, 6], the language of well-balanced parenthesis, $\mathcal{L}(G')$ contains the set of all parentheses strings with at most k levels of nesting. Note that a string consisting of n open parenthesis followed by n matching closed parenthesis has zero levels of nesting, whereas the string "((())())" has one level. As an another example, consider RNA-folding [8, 39, 44] which is a basic problem in computational biology and can be modeled by grammars. The restricted language $\mathcal{L}(G')$ for RNA-folding denotes the set of all RNA strings with at most k nested folds. In typical applications, we do not expect the number of nested non-linear productions used in the derivation of a valid string to be too large [14, 23, 4].

Among our other results, we consider exact algorithms for several other notable sub-classes of the CFG's. In particular, we develop exact quadratic time language edit distance algorithms for the linear, metalinear, and superlinear languages. Moreover, we show matching lower bound assuming the Strong Exponential Time Hypothesis [18, 19]. The figure to the right displays the hierarchical relationship between these grammars, where all upwards lines denote strict containment. Interestingly, till date there exists no parsing algorithm for the ultralinear grammars that runs in time $o(n^{\omega})$, while a $O(n^2)$ algorithm exists for the metalinear grammars. In addition, there is no combinatorial algorithm that runs in $o(n^3)$ time. In this paper, we derive conditional lower bound exhibiting why a faster algorithm has so far been elusive for the ultralinear grammars, clearly demarking the boundary where parsing becomes hard!

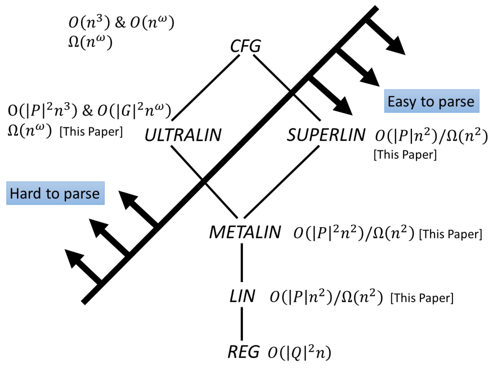
$O(n^3)$ & $O(n^{\omega})$
$\Omega(n^{\omega})$
CFG
$O(|P|^2n^3)$ & $O(|G|^2n^{\omega})$
$\Omega(n^{\omega})$ [This Paper]
ULTRALIN
SUPERLIN $O(|P|n^2)/\Omega(n^2)$
[This Paper]
Easy to parse
Hard to parse
METALIN $O(|P|^2n^2)/\Omega(n^2)$ [This Paper]
LIN $O(|P|n^2)/\Omega(n^2)$ [This Paper]
REG $O(|Q|^2n)$

Figure 1 CFG Hierarchy: Upper bounds shown first followed by lower bounds for each class of grammars. Here $|P|$ is the number of productions in the grammar [38] [1] [40].

1.1 Results & Techniques

Lower Bounds. Our first hardness result is a lower bound for the problem of linear language edit distance. We show that a truly subquadratic time algorithm for linear language edit distance would refute the Strong Exponential Time Hypothesis (SETH). This further builds on a growing family of "SETH-hard" problems – those for which lower bounds can be proven conditioned on SETH. We prove this result by reducing binary string edit distance, which has been shown to be SETH-hard [9, 5], to linear language edit distance.

▶ Theorem (Linear Grammar Hardness of Parsing). There exists no algorithm to compute the minimum edit distance between a string $\overline{x}$, $|\overline{x}| = n$, and a linear language $\mathcal{L}(G)$ in $o(n^{2-\epsilon})$ time for any constant $\epsilon > 0$, unless SETH is false.

Our second, and primary hardness contribution is a conditional lower bound on the recognition problem for ulralinear languages. Our result builds closely off of the work of Abboud, Backurs and V. Williams [1], who demonstrate that finding an $o(n^3)$-time combinatorial algorithm or any $o(n^{\omega})$-algorithm for context free language recognition would result in faster algorithms for the k-clique problem and falsify a well-known conjecture in graph algorithms. We modify the grammar in their construction to be ultralinear, and then demonstrate that the same hardness result holds for our grammar. See the full version for details.

▶ Theorem (Ultralinear Grammar Hardness of Parsing). There is a ultralinear grammar $\mathcal{G}_U$ such that if we can solve the membership problem for a string of length n in time $O(|\mathcal{G}_U|^{\alpha} n^c)$ for any fixed constant $\alpha > 0$, then we can solve the $3k$-clique problem on a graph with n nodes in time $O(n^{c(k+3)+3\alpha})$.

Upper Bounds. We provide the first quadratic time algorithms for linear (Theorem 7), superlinear (in full version), and metalinear language edit distance (in full version), running in $O(|P|n^2)$, $O(|P|n^2)$ and $O(|P|^2n^2)$ time respectively. This exhibits a large family of grammars for which edit distance computation can be done faster than for general context free grammars, as well as for other well known grammars such as the Dyck grammar [1]. Along with our lower bound for the ultralinear language parsing, this demonstrates a clear division between those grammars for which edit distance can be efficiently calculated, and those for which the problem is likely to be fundamentally hard. Our algorithms build progressively off the construction of a *linear language edit distance graph*, reducing the problem of edit distance computation to computing shortest path on a graph with $O(|P|n^2)$ edges (Section 2).

Our main contribution is an additive approximation for language edit distance. We first present a cubic time exact algorithm, and then show a general procedure for modifying this algorithm, equivalent to forgetting states of the underlying dynamic programming table, into a family of *amnesic* dynamic programming algorithms. This produces *additive approximations* of the edit distance, and also provides a tool for proving general bounds on any such algorithm. In particular, we provide two explicit procedures for forgetting dynamic programming states: *uniform* and *non-uniform* grid approximations achieving the following approximation-running time trade-off. See Section 4, and the full version for missing proofs.

▶ **Theorem 4.** *If $\mathcal{A}$ is a γ-uniform grid approximation, then the edit distance computed by $\mathcal{A}$ satisfies $|OPT| \leq |\mathcal{A}| \leq |OPT| + O(k^*\gamma)$ and it runs in $O(|P|(n^2 + (\frac{n}{\gamma})^3))$ time.*

▶ **Theorem 5.** *Let $\mathcal{A}$ be any γ-non-uniform grid approximation, then the edit distance computed by $\mathcal{A}$ satisfies $|OPT| \leq |\mathcal{A}| \leq |OPT| + O(k\gamma)$and it runs in $O\big(|P|\big(n^2 + \frac{n^3}{\gamma^2}\big)\big)$ time.*

We believe that our amnesic technique can be applied to wide range of potential dynamic programming approximate algorithms, and lends itself particularly well to randomization.

2 Linear Grammar Edit Distance in Quadratic Time

We first introduce a graph-based exact algorithm for linear grammar, that is a grammar $G = (Q, \Sigma, P, S)$ where every production has one of the following forms: $A \to \alpha B$, $A \to B\alpha$, $A \to \alpha B \beta$, or $A \to \alpha$ where $A, B \in Q$, and $\alpha, \beta \in \Sigma$. Given G and a string $\overline{x} = x_1x_2 \dots x_n \in \Sigma^*$, we give an $O(|P|n^2)$ algorithm to compute edit distance between $\overline{x}$ and G in this section. The algorithm serves as a building block for the rest of the paper.

Note that if we only have productions of the form $A \to \alpha B$ (or $A \to B\alpha$ but not both) then the corresponding language is regular, and all regular languages can be generated in this manner. However, there are linear languages that are not regular. For instance, the language $\{0^n1^n \mid n \in \mathbb{N}\}$ can be produced by the linear grammar $S \to 0S1 \mid \epsilon$, but cannot be produced by any regular grammar [37]. Therefore, regular languages are a strict subclass of linear languages. Being a natural extension of the regular languages, the properties and applications of linear languages are of much interest[13, 36].

Algorithm. Given inputs G and $\overline{x}$, we construct a weighted digraph $\mathcal{T} = \mathcal{T}(G, \overline{x})$ with a designated vertex $S^{1,n}$ as the source and t as the sink such that the weight of the shortest path between them will be the minimum language edit distance of $\overline{x}$ to G.

Construction. The vertices of $\mathcal{T}$ consist of $\binom{n}{2}$ *clouds*, each corresponding to a unique substring of $\overline{x}$. We use the notation (i, j) to represent the cloud, $1 \leq i \leq j \leq n$, corresponding to the substring $x_i x_{i+1} \ldots x_j$. Each cloud will contain a vertex for every nonterminal in Q. Label the nonterminals $Q = \{S = A_1, A_2, \ldots, A_q\}$ where $|Q| = q$, then we denote the vertex corresponding to A_k in cloud (i, j) by $A_k^{i,j}$. We will add a new sink node t, and use $S^{1,n}$ as the source node s. Thus the vertex set of $\mathcal{T}$ is $V(\mathcal{T}) = \{A_k^{i,j} \mid 1 \leq i \leq j \leq n,\ 1 \leq k \leq q\} \cup \{t\}$. The edges of $\mathcal{T}$ will correspond to the productions in G. Each path from a nonterminal $A_k^{i,j}$ in (i, j) to t corresponds to the production of a legal string w, that is a string that can be derived starting from A_k and following the productions of P, and a sequence of editing procedures to edit w to $x_i x_{i+1} \ldots x_j$. For any cloud (i, j), edges will exist between two nonterminals in (i, j), and from nonterminals in (i, j) to nonterminals in $(i + 1, j)$ and $(i, j - 1)$. Our goal will be to find the shortest path from $S^{1,n}$, the starting nonterminal S in cloud $(1, n)$, to the sink t.

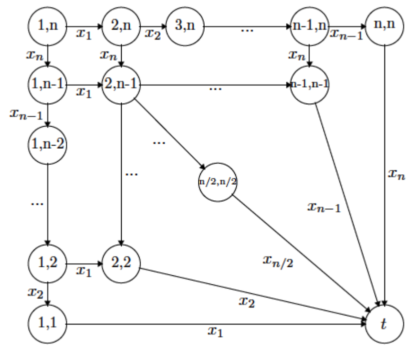

Figure 2 Clouds corresponding to Linear Grammar Edit Distance Graph Construction. Each cloud contains a vertex for every nonterminal

Adding the edges. Each edge in $\mathcal{T}$ is directed, has a weight in $\mathbb{Z}^+$ and a label from $\{x_1, x_2, .., x_n, \epsilon\} \cup \{\epsilon(\alpha) \mid \alpha \in \Sigma\}$, where $\epsilon(\alpha)$ corresponds to the deletion of α. If u, v are two vertices in $\mathcal{T}$, then we use the notation $u \xrightarrow[w(u,v)]{\ell} v$ to denote the existence of an edge from u to v with weight $w(u, v)$ and edge label ℓ. For any nonterminal $A \in Q$, define $null(A)$ to be the length of the shortest string in Σ^* derivable from A, which can be precomputed in $O(|Q||P|\log(|Q|))$ time for all $A \in Q$ (see full version for details). This is the minimum cost of deleting a whole string produced by A. Given input $x_1 x_2 \ldots x_n$, for all nonterminals A_k, A_r and every $1 \leq i \leq j \leq n$, the construction is as follows:

- **Legal Productions:** For $i \neq j$, then if $A_k \rightarrow x_i A_r$ is a production, add the edge $A_k^{i,j} \xrightarrow[0]{x_i} A_r^{i+1,j}$ to $\mathcal{T}$. If $A_k \rightarrow A_r x_j$ is a production, add the edge $A_k^{i,j} \xrightarrow[0]{x_j} A_r^{i,j-1}$ to $\mathcal{T}$.
- **Completing Productions:** If $A_k \rightarrow x_i$ is a production, add the edge $A_k^{i,i} \xrightarrow[0]{x_i} t$ to $\mathcal{T}$. If $A_k \rightarrow x_i A_r$ or $A_k \rightarrow A_r x_i$ is a production, add the edge $A_k^{i,i} \xrightarrow[null(A_r)]{x_i} t$ to $\mathcal{T}$.
- **Insertion:** If $A_k \rightarrow x_i A_k$ is *not* a production, add the edge $A_k^{i,j} \xrightarrow[1]{x_i} A_k^{i+1,j}$ to $\mathcal{T}$. If $A_k \rightarrow A_k x_j$ is *not* a production, add $A_k^{i,j} \xrightarrow[1]{x_j} A_k^{i,j-1}$. *{these are called insertion edges.}*
- **Deletion:** For every production $A_k \rightarrow \alpha A_r$ or $A_k \rightarrow A_r \alpha$, add the edge $A_k^{i,j} \xrightarrow[1]{\epsilon(\alpha)} A_r^{i,j}$. *{these are called deletion edges.}*
- **Replacement:** For every production $A_k \rightarrow \alpha A_r$, if $\alpha \neq x_i$, then add the edge $A_k^{i,j} \xrightarrow[1]{x_i} A_r^{i+1,j}$ to T. For every production $A_k \rightarrow A_r \alpha$, if $\alpha \neq x_j$, add $A_k^{i,j} \xrightarrow[1]{x_j} A_r^{i,j-1}$ to $\mathcal{T}$. For any A_k such that $A_k \rightarrow x_i$ is not a production, but $A_k \rightarrow \alpha$ is a production with $\alpha \in \Sigma$, add the edge $A_k^{i,i} \xrightarrow[1]{x_i} t$ to $\mathcal{T}$.*{these are called substitution or replacement edges.}*

▶ **Theorem 6.** *For every $A_k \in Q$ and every $1 \leq i \leq j \leq n$, the cost of the shortest path of from $A_k^{i,j}$ to the sink $t \in \mathcal{T}$ is d if and only if d is the minimum edit distance between the* *string $x_i \dots x_j$ and the set of strings which can be derived from A_k.*

▶ **Theorem 7.** *The cost of the shortest path from $S^{1,n}$ to t in the graph $\mathcal{T}$ is the minimum edit distance which can be computed in $O(|P|n^2)$ time.*

3 Context Free Language Edit Distance

In this section, we develop an exact algorithm which utilizes the graph construction presented in Section 2 to compute the language edit distance of a string $\overline{x} = x_1 \dots x_n$ to any context free grammar (CFG) $G = (Q, \Sigma, P, S)$. We use a standard normal form for G, which is Chomsky normal form except we also allow productions of the form $A \to Aa|aA$, where $A \in Q, a \in \Sigma$. For us, the important property of this normal form is that every non-linear production must be of the form $A \to BC$, with exactly two non-terminals on the right hand side. Any CFG can be reduced to this normal form (see full version for more details).

Let $P_L, P_{NL} \subset P$ be the subsets of (legal) linear and non-linear productions respectively. Then for any nonterminal $A \in Q$, the grammar $G_L = (Q, \Sigma, P_L, A)$ is linear, and we denote the corresponding linear language edit distance graph by $\mathcal{T}(G_L, \overline{x}) = \mathcal{T}$, as constructed in Section 2. Let L_i be the set of clouds in $\mathcal{T}$ which correspond to substrings of length i (so $L_i = \{(k,j) \in \mathcal{T} \mid j - k + 1 = i\}$). Then $L_1, \dots, L_n$ is a *layered partition* of $\mathcal{T}$. Let t be the sink of $\mathcal{T}$. We write $\mathcal{T}^R$ to denote the graph $\mathcal{T}$ where the direction of each edge is reversed. Let L_i^R denote the edge reversed subgraph of L_i. In other words, L_i^R is the subgraph of $\mathcal{T}^R$ with the same vertex set as L_i. Our algorithm will add some additional edges within L_i^R, and some additional edges from t to L_i^R, for all $1 \le i \le n$, resulting in an augmented subgraph which we denote $\overline{L}_i^R$. We then compute single source shortest path from t to $\overline{L}_i^R \cup \{t\}$ in phase i. Our algorithm will maintain the property that, after phase $q - p + 1$, if $A^{p,q}$ is any nonterminal in cloud (p,q) then the weight of the shortest path from t to $A^{p,q}$ is precisely the minimum edit distance between the string $x_p x_{p+1} \dots x_q$ and the set of strings that are legally derivable from A. The algorithm is as follows:

Algorithm: Context Free-Exact

1. **Base Case: strings of length** 1. For every non-linear production $A \to BC$, and every $1 \le \ell \le n$, add the edges $A^{\ell,\ell} \xleftarrow[null(B)]{} C^{\ell,\ell}$ and $A^{\ell,\ell} \xleftarrow[null(C)]{} B^{\ell,\ell}$ to L_1^R. Note that the direction of the edges are reversed because we are adding edges to L_1^R and not L_1. Call the resulting augmented graph $\overline{L_1^R}$.
2. Solve single source shortest path from t to every vertex in $\overline{L_1^R} \cup \{t\}$. Store the value of the shortest path from t to every vertex in $\overline{L_1^R}$, and an encoding of the path itself. For any $1 \le p \le q \le n$ and $A^{p,q} \in L_{q-p+1}$, we write $T_{p,q}(A)$ to denote the weight of the shortest path from t to $A^{p,q}$. After having computed shortest paths from t to every vertex in the subgraphs $\overline{L}_1^R, \dots, \overline{L}_{i-1}^R$, we now consider L_i^R.
3. **Induction: strings of length** i**.** For every edge from a vertex $A^{p,q}$ in L_i to a vertex $B^{p+1,q}$ or $B^{p,q-1}$ in L_{i-1} with cost $\gamma \in \{0,1\}$, add an edge from t to $A^{p,q} \in L_i^R$ with cost $T_{p+1,q}(B) + \gamma$ or $T_{p,q-1}(B) + \gamma$, respectively. These are the linear production edges created in the linear grammar edit distance algorithm.
4. For every non-linear production $A \to BC$ and every vertex $A^{p,q} \in L_i^R$, add an edge from t to $A^{p,q}$ in L_i^R with cost c where $c = \min_{p \le \ell < q} T_{p,\ell}(B) + T_{\ell+1,q}(C)$. The indices $p \le \ell < q$ are called *splitting points,* as they specify where the string $x_p, \dots, x_q$ is split by the production $A \to BC$. To later recover the derivation, we store the specific ℓ which yields the minimum value of the above equation.

5. For every non-linear production $A \to BC$, add the edge $A^{p,q} \xleftarrow[null(B)]{} C^{p,q}$ and $A^{p,q} \xleftarrow[null(C)]{} B^{p,q}$ to L_i^R.
6. After adding the edges in steps 3-5, we call the resulting graph $\overline{L}_i^R$. Then compute shortest path from t to every vertex in the subgraph $\overline{L}_i^R \cup \{t\}$, and store the values of the shortest paths, along with an encoding of the paths themselves.
7. Repeat for $i = 1, 2, \ldots, n$. Return the value $T_{1,n}(S)$.

▶ **Theorem 8.** *For any nonterminal $A \in Q$ and $1 \le p \le q \le n$, the weight of the shortest path from $A^{p,q} \in \overline{L}_i$ to t is the minimum edit distance between the substring $x_p \ldots x_q$ and the set of strings which can be legally produced from A, and the overall time required to compute the language edit distance is $O(|P|n^3)$.*

4 Context Free Language Edit Distance Approximation

Now this cubic time algorithm itself is not an improvement on that of Aho and Peterson [3]. However, by strategically modifying the construction of the subgraphs L_i by **forgetting** to compute some of the non-linear edge weights (and taking the minimum over fewer splitting points for those that we do compute), we can obtain an additive approximation of the minimum edit distance. We introduce a family of approximation algorithms which do just this, and prove a strong general bound on their behavior. Our results give bounds for the performance of our algorithm for any CFG. Additionally, for any k-ultralinear language, our results also give explicit $O(k\sqrt{n})$ and $O(2^k n^{1/3})$ additive approximations from this family which run in quadratic time. Note that, as shown in our construction in the proof of hardness of parsing ultralinear grammars, for any k we can restrict any context free grammar G to a k-ultralinear grammar G' such that $\mathcal{L}(G') \subseteq \mathcal{L}(G)$ contains all words that can be derived using fewer than $\le k$ nested non-linear productions (see full version for a more formal definition of k and hardness proofs).

▶ **Definition 9.** For any Context Free Language edit distance approximation algorithm $\mathcal{A}$, we say that $\mathcal{A}$ is in the family $\mathcal{F}$ if it follows the same procedure as in the exact algorithm with the following modifications:

1. **Subset of non-linear productions.** $\mathcal{A}$ constructs the non-linear production edges in step 4 for the vertices in some subset of the total set of clouds $\{(p,q) \mid 1 \le p \le q \le n\}$.
2. **Subset of splitting points.** For every cloud (p,q) that $\mathcal{A}$ computes non-linear production edges for, in step 4 of the algorithm when computing the weight c of any edge in this cloud it takes minimum over only a subset of the possible splitting points $p, \ldots, q$ (where this subset is the same for every non-linear edge weight computed in (p,q)).

By forgetting to construct all non-linear production edges, and by taking a minimum over fewer values when we do construct non-linear production edges, the time taken by our algorithm to construct new edges can be substantially reduced. Roughly, the intuition for how we can still obtain an additive approximation is as follows. If the shortest path to the sink in the exact algorithm uses a non-linear edge from a vertex $A^{p,q}$ in cloud (p,q), then naturally our approximation algorithm would also use such an edge if it existed. However, it is possible that nonlinear edges were not constructed for cloud (p,q) by the approximation. Still, what we can do is find the closest cloud (p',q'), with $p \le p' \le q' \le q$, such that nonlinear edges were constructed in (p',q'), and then follow the insertion edges $A^{p,q} \to A^{p+1,q} \to \cdots \to A^{p',q'}$, and take the desired non-linear production edge from $A^{p',q'}$. The additional incurred cost is at most $|p-p'|+|q-q'|$, or the distance to the nearest cloud with non-linear edges, and this cost is incurred at most once for every non-linear production in an optimal derivation.

We now give two explicit examples of how steps 1 and 2 can be implemented. We later prove explicit bounds on the approximations of these examples in Theorems 4 and 5. In both examples a *sensitivity parameter*, γ, is first chosen. We use $|OPT|$ to denote the optimum language edit distance, and $|\mathcal{A}|$ to denote the edit distance computed by an approximation algorithm $\mathcal{A}$.

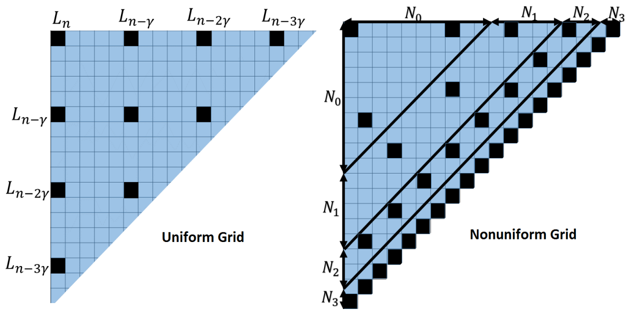

Figure 3 Non-uniform edges are computed only for a subset of the clouds (colored black). Only a subset of the splitting points are considered while computing the weights.

▶ **Definition 10.** An approximation algorithm $\mathcal{A} \in \mathcal{F}$ is a γ-*uniform grid* approximation if for $i = n, (n-\gamma), (n-2\gamma), \dots, (n - \lfloor \frac{n}{\gamma} \rfloor \gamma)$ (see Figure 3).

1. $\mathcal{A}$ constructs non-linear production edges only for an evenly-spaced $1/\gamma$ fraction of the clouds in L_i, and no others, where γ is a specified sensitivity parameter.
2. Furthermore, for every non-linear edge constructed, $\mathcal{A}$ considers only an evenly-spaced $1/\gamma$ fraction of the possible break points.

Here if i or $(n-i+1)$ (the number of substrings of length i) is not evenly divisible by γ, we evenly space the clouds/breakpoints until no more will fit.

We will later see that the running time of such a γ-uniform grid approximation is $O(|P|(n^2 + (\frac{n}{\gamma})^3))$, and in particular for any k-ultralinear grammar G it gives an additive approximation of $O(2^k\gamma)$. Thus by setting $\gamma = n^{1/3}$, we get an $O(2^k n^{1/3})$-approximation in $O(|P|n^2)$ time (Theorem 4).

▶ **Definition 11.** For $i = 0, 1, \dots, \log(n)$, set $N_i = \{L_j \mid \frac{n}{2^{i+1}} < j \leq \frac{n}{2^i}\}$. Let $N_i' \subset N_i$ be an evenly-spaced $\min\{\frac{2^i}{\gamma}, 1\}$-fraction of the L_j's in N_i (subset of diagonals). Then, an approximation algorithm $\mathcal{A} \in \mathcal{F}$ is a γ-*non-uniform grid* approximation if, for every $L_j \in N_i'$, $\mathcal{A}$ computes non-linear production edges only for a $\min\{\frac{2^i}{\gamma}, 1\}$-evenly-spaced fraction of the clouds in L_j. Furthermore, for any of these clouds in N_i' for which $\mathcal{A}$ does compute non-linear production edges, $\mathcal{A}$ considers only an evenly-spaced $\min\{\frac{2^i}{\gamma}, 1\}$ -fraction of all possible break points (see Figure 3 (right)).

We will see that the running time of a γ-non-uniform grid approximation is $O(|P|(n^2 + \frac{n^3}{\gamma^2}))$, and in particular for any k-ultralinear grammar, or if k is the maximum number of nested non-linear productions, it gives an additive approximation of $O(k\gamma)$. Hence setting $\gamma = \sqrt{n}$, we get an additive approximation of $O(k\sqrt{n})$ in quadratic time (Theorem 5).

4.1 Analysis.

The rest of this section will be devoted to proving bounds on the performance of approximation algorithms in $\mathcal{F}$. We use $\mathcal{T}^{OPT}$ to denote the graph which results from adding all the edges specified in the exact algorithm to $\mathcal{T}$. Recall that $\mathcal{T}$ is the graph constructed from the linear productions in G. For $\mathcal{A} \in \mathcal{F}$, we write $\mathcal{T}^{\mathcal{A}}$ to denote the graph which results from adding the edges specified by the approximation algorithm $\mathcal{A}$. Note that since $\mathcal{A}$ functions by forgetting to construct a subset of the non-linear edges created by the exact algorithm, we have that the edge sets satisfy $E(\mathcal{T}) \subseteq E(\mathcal{T}^{\mathcal{A}}) \subseteq E(\mathcal{T}^{OPT})$. We now introduce the primary structure which will allow us to analyze the execution of our language edit distance algorithms.

Binary Production-Edit Trees.

▶ **Definition 12.** A *production-edit tree* (PET) $\mathbb{T}$ for grammar G and input string $\overline{x}$ is a binary tree which satisfies the following properties:

1. Each node of $\mathbb{T}$ stores a path in the linear grammar edit distance graph $\mathcal{T} = \mathcal{T}(G_L, \overline{x})$ (see Section 2 and 3). The path given by the root of $\mathbb{T}$ must start at the source vertex $S^{1,n}$ of $\mathcal{T}$.
2. For any node $v \in \mathbb{T}$, let $A^{p,q}, B^{r,s}$ be the starting and ending vertices of the corresponding path. If $B^{r,s}$ is not the sink t of $\mathcal{T}$, then v must have two children, v_R, v_L, such that there exists a production $B \to CD$ and the starting vertices of the paths in v_L and v_R are $C^{r,\ell}$ and $D^{\ell+1,s}$ respectively, where ℓ is some splitting point $r - 1 \leq \ell \leq s$. If $\ell = r - 1$ or $\ell = s$, then one of the children will be in the same cloud (r, s) as the ending cloud of the path given by v, and the other will be called a *nullified node.* This corresponds to the case where one of the *null* edges created in step 5 of the exact algorithm is taken.
3. If the path in $v \in \mathbb{T}$ ends at the sink of $\mathcal{T}$, then v must be a leaf in $\mathbb{T}$. If $A^{p,q}$ is the starting vertex of the path, this means that the path derives the entire substring $x_p \dots x_q$ using only linear productions. Thus a node v is a leaf of $\mathbb{T}$ if and only if it either ends at the sink or is a nullified node. It follows from 2. and 3. that every non-leaf node has exactly 2 children.

Notation: To represent a node in $\mathbb{T}$ that is a path of cost c from $A^{p,q}$ to either $B^{r,s}$, or t, we will use the notation $[A^{p,q}, B^{r,s}, c]$, or $[A^{p,q}, t, c]$, respectively. If one of the arguments is either unknown or irrelevant, we write $\cdot$ as a placeholder. In the case of a nullified node, corresponding to the nullification of $A \in Q$, we write $[A, t, null(A)]$ to denote the node. Note, since we are now dealing with two *types* of graphs, to avoid confusion whenever we are talking about a vertex $A^{p,q}$ in any of the edit-distance graphs (such as $\mathcal{T}, \mathcal{T}^{\mathcal{A}}, \mathcal{T}^{OPT}$, ect), we will use the term *vertex.* When referring to the elements of a PET $\mathbb{T}$ we will use the term *node.* Also note that all error productions are linear.

We can now represent any sequence of edits produced by a language edit distance algorithm by such a PET, where the edit distance is given by the sum of the costs stored in the nodes of the tree. To be precise, if $[\cdot, \cdot, c_1], \dots, [\cdot, \cdot, c_k]$ is the set of all nodes in $\mathbb{T}$, then the associated total cost $\|\mathbb{T}\| = \sum_{i=1}^{k} c_i$. Let $\mathcal{D}_{\mathcal{A}}$ be the set of PET's $\mathbb{T}$ compatible with a fixed approximation algorithm $\mathcal{A} \in \mathcal{F}$.

▶ **Definition 13** (PET's compatible with $\mathcal{A}$)**.** For an approximation algorithm $\mathcal{A} \in \mathcal{F}$, let $\mathcal{D}_{\mathcal{A}} \subset \mathcal{F}$ be the set of PET's $\mathbb{T}$ which satisfy the following constraints:

1. If $[A^{p,q}, B^{r,s}, \cdot]$ is a node in $\mathbb{T}$, where $A, B \in Q$, then $\mathcal{A}$ must compute non-linear edges for the cloud $(r, s) \in \mathcal{T}^{\mathcal{A}}$.
2. If $[C^{r,\ell}, \cdot, \cdot], [D^{\ell+1,s}, \cdot, \cdot]$ are the left and right children of a node $[A^{p,q}, B^{r,s}, \cdot]$ respectively, then $\mathcal{A}$ must consider the splitting point $\ell \in [p, q)$ when computing the weights of the non-linear edges in the cloud $(r, s) \in \mathcal{T}^{\mathcal{A}}$.

The set $\mathcal{D}_{\mathcal{A}}$ is then the set of all PET's which utilize only the non-linear productions and splitting points which correspond to edges that are actually constructed by the approximation algorithm $\mathcal{A}$ in $\mathcal{T}^{\mathcal{A}}$. Upon termination, any $\mathcal{A} \in \mathcal{F}$ will return the value $\|\mathbb{T}_{\mathcal{A}}\|$ where $\mathbb{T}_{\mathcal{A}} \in \mathcal{D}_{\mathcal{A}}$ is the tree corresponding to the shortest path from t to $S^{1,n}$ in $\mathcal{T}^{\mathcal{A}}$. The following theorem is not difficult to show.

▶ **Theorem 14.** *Fix any $\mathcal{A} \in \mathcal{F}$, and let c be the edit distance returned after running the approximation algorithm $\mathcal{A}$. Then if $\mathbb{T}$ is any PET in $\mathcal{D}_{\mathcal{A}}$, we have $c \leq \|\mathbb{T}\|$.*

Note that since the edges of $\mathcal{T}^{\mathcal{A}}$ are a subset of the edges of $\mathcal{T}^{OPT}$ considered by an exact algorithm OPT, we also have $c \geq \|\mathbb{T}_{OPT}\|$, where $\mathbb{T}_{OPT}$ is the PET given by the exact algorithm. To prove an upper bound on c, it then suffices to construct a explicit $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$, and put a bound on the size of $\|\mathbb{T}\|$. Thus, in the remainder of our analysis our goal will be to construct such a $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$. We now introduce our precision functions.

▶ **Definition 15** (Precision Functions)**.** For any cloud $(p, q) \in \mathcal{T}^{\mathcal{A}}$, let $\alpha(p, q)$ be any upper bound on the minimum distance $d^*((p, q), (r, s)) = (r - p) + (q - s)$ such that $p \leq r \leq s \leq q$ and $\mathcal{A}$ computes non-linear edge weights for the cloud (r, s) . Let $\beta(p, q)$ be an upper bound on the maximum distance between any two splitting points which are considered by $\mathcal{A}$ in the construction of the non-linear production edges originating in a cloud (r, s) such that $\mathcal{A}$ computes non-linear edge weights for (r, s) and $d^*((p, q), (r, s)) \leq \alpha(p, q)$. Furthermore, the precision functions must satisfy $\alpha(p, q) \geq \alpha(p', q')$ and $\beta(p, q) \geq \beta(p', q')$ whenever $(q - p) \geq (q' - p')$.

While the approximation algorithms presented in this paper are deterministic, the definitions of $\alpha(p, q)$ and $\beta(p, q)$ allow the remaining theorems to be easily adapted to algorithms which *randomly forget* to compute non-linear edges. While our paper considers only two explicit approximation algorithms, stating our results in this full generality substantially easies the analysis. Both Theorems 4 and 5 will follow easily once general bounds are proven, and without the generality two distinct proofs would be necessary.

Constructing a PET $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$ similar to $\mathbb{T}_{OPT}$.

Our goal is now to construct a PET $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$ with bounded cost. We do this by considering each node v of $\mathbb{T}_{OPT}$ and constructing a corresponding node u in $\mathbb{T}$ such that the path stored in u *imitates* the path in v as closely as possible. A perfect imitation may not be feasible if the path at v uses a non-linear production edge in a cloud that $\mathcal{A}$ does not compute non-linear edges for. Whenever this happens, we will need to move to the closest cloud which $\mathcal{A}$ does consider before making the *same* non-linear production that the exact algorithm did. Afterwards, the ending cloud of our path will deviate from that of the optimal, so we will need to bound the total deviation that can occur throughout the construction of our tree in terms of $\alpha(p, q)$ and $\beta(p, q)$. The following lemma will be used crucially in this regard for the proof of our construction in Theorem 17. The lemma takes as input a node $[A^{p,q}, B^{r,s}, c] \in \mathbb{T}_{OPT}$ and a cloud (p', q') such that $x_p, \ldots, x_q$ is not disjoint from $x_{p'}, \ldots, x_{q'}$, and constructs a path $[A^{p',q'}, B^{r',s'}, c']$ of bounded cost that is compatible with a PET $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$.

▶ **Lemma 16.** *Let $[A^{p,q}, B^{r,s}, c]$ be any non-leaf node in $\mathbb{T}_{OPT}$, and let $\mathcal{A} \in \mathcal{F}$ be an approximation algorithm with precision functions $\alpha(p, q), \beta(p, q)$. If p', q' satisfy $p \leq q'$ and $p' \leq q$, then there is a path from $A^{p',q'}$ to $B^{r',s'}$, where $r \leq r' \leq s' \leq s$, of cost $c' \leq c + (|p' - p| + |q' - q|) - (|r' - r| + |s' - s|) + 2\alpha(r, s)$ such that $\mathcal{A}$ computes non-linear production edges for cloud (r', s'). Furthermore, for any leaf node $[A^{p,q}, t, c] \in \mathbb{T}_{OPT}$, we can construct a path from $A^{p',q'}$ of cost at most $c' \leq c + (|p' - p| + |q' - q|)$ to the sink.*

We will now iteratively apply Lemma 16 to each node $v \in \mathbb{T}_{OPT}$ from the root down, transforming it into a new node $\psi(v) \in \mathbb{T}$. Here ψ will be a surjective function $\psi : V(\mathbb{T}_{OPT}) \to V(\mathbb{T})$. Lemma 16 will guarantee that the cost of $\psi(v)$ is not too much greater than that of v. If during the construction of $\mathbb{T}$, the substrings corresponding to v and $\psi(v)$ become disjoint, then we will transform the *entire* subtree rooted at v into a single node $\psi(v) \in \mathbb{T}$, thus the function may not be injective.

Let v be any node in $\mathbb{T}_{OPT}$, and u its parent node if v is not the root. Let $(p, q), (r, s) \in \mathcal{T}$ and $(p_v, q_v), (r_v, s_v) \in \mathcal{T}$ be the starting and ending clouds of u and v respectively. Similarly let $(p', q'), (r', s')$ and $(p'_v, q'_v), (r'_v, s'_v)$ be the starting and ending clouds of $\psi(u)$ and $\psi(v)$ respectively. Furthermore, let (p_X, q_X) and (p'_X, q'_X), where $X = L$ for left child and $X = R$ for right child, be the starting clouds of the left and right children of u and $\psi(u)$ respectively. Let c_v be the cost of v, and let $\overline{c_v}$ be the cost of v plus the cost of all the descendants of v. Finally, let c'_v be the cost of $\psi(v)$. An abbreviated version of Theorem 17 (see the full version for extended statement) relates the cost of v with $\psi(v)$ in terms of the starting and ending clouds by defining ψ inductively from the root of $\mathbb{T}_{OPT}$ down. The theorem uses Lemma 16 repeatedly to construct the nodes of $\mathbb{T}$.

▶ **Theorem 17.** *For any approximation algorithm $\mathcal{A} \in \mathcal{F}$ with precision functions α, β, there exists a PET $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$ and a PET mapping $\psi : V(\mathbb{T}_{OPT}) \to V(\mathbb{T})$ such that $\mathbb{T}_{OPT}$ can be partitioned into disjoint sets $U^1_{NL} \cup U^2_{NL} \cup U^1_L \cup U^2_L \cup X$ with the following properties. For $v \in \mathbb{T}_{OPT}$, if $v \in U^1_{NL} \cup U^2_{NL}$ then v satisfies (Non-leaf), and if $v \in U^1_L \cup U^2_L$ then v satisfies (Leaf):*

$$c'_v \le c_v + \big(|p'_v - p_v| + |q'_v - q_v|\big) - \big(|r'_v - r_v| + |s'_v - s_v|\big) + 2\alpha(r_v, s_v) \qquad \text{(Non-leaf)}$$
$$c'_v \le \overline{c_v} + |p'_v - p_v| + |q'_v - q_v| + \beta(r, s) \qquad \text{(Leaf)}$$

Furthermore $(|p'_L - p_L| + |q'_L - q_L|) + (|p'_R - p_R| + |q'_R - q_R|) \le |r' - r| + |s' - s| + 2\beta(r, s)$ $(*)$

Let $\mathbb{T}'_{OPT} \subset \mathbb{T}_{OPT}$ be the subgraph of nodes v in the tree for which either v is the only node mapped to $\psi(v) \in \mathbb{T}$, or v is the node closest to the root that is mapped to $\psi(v)$. In the previous theorem, the set X corresponds to the nodes v for which $\psi(v) = \psi(u)$ such that u is an ancestor of v in $\mathbb{T}_{OPT}$. So $\mathbb{T}'_{OPT} = \mathbb{T}_{OPT} \setminus X$. The final theorem is the result of summing over the bounds from Theorem 17 for all $v_j \in \mathbb{T}'_{OPT}$, applying the appropriate bound depending on the set v_j belongs to.

▶ **Theorem 18.** *For any $\mathcal{A} \in \mathcal{F}$ with precision functions α, β, let $\mathbb{T}_{OPT}$ be the PET of any optimal algorithm. Label the nodes of $\mathbb{T}'_{OPT} \subset \mathbb{T}_{OPT}$ by $v_1 \dots v_K$. For $1 \le i \le K$, let $(p_i, q_i), (r_i, s_i)$ be the starting and ending clouds of the path v_i in $\mathcal{T}$, then*

$$|OPT| \le |\mathcal{A}| \le |OPT| + \sum_{v_j \in \mathbb{T}'_{OPT}} \Big(2\alpha(r_j, s_j) + 3\beta(r_j, s_j)\Big)$$

As an illustration of Theorem 18, consider the γ-uniform grid approximation of Theorem 4. In this case, we have the upper bound $\alpha(r_j, s_j) = \beta(r_j, s_j) = 2\gamma$ for all $v_j \in \mathbb{T}_{OPT}$. Since there are k^* total vertices in $\mathbb{T}_{OPT}$, we get $|OPT| \le |\mathcal{A}| \le |OPT| + 10\gamma k^*$. To analyze the running time, note that we only compute non-linear production edges for $(n/\gamma)^2$ clouds, and in each cloud that we compute non-liner edges for we consider at most n/γ break-points. Thus the total runtime is $O(|P|(\frac{n}{\gamma})^3)$ to compute non-linear edges, and $O(|P|n^2)$ to a shortest path algorithm on $\mathcal{T}^{\mathcal{A}}$, for a total runtime of $O(|P|(n^2 + (\frac{n}{\gamma})^3))$.

Our second illustration of Theorem 18 is the γ-non-uniform grid approximation of Theorem 5. Here we obtain a $O(k\gamma)$ additive approximation in time $O(|P|(n^2 + \frac{n^3}{\gamma^2}))$. A detailed analysis can be found in the full version.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is valiant's parser. FOCS, 2015.

2 Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

3 Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.*, 1(4), 1972.

4 Rolf Backofen, Dekel Tsur, Shay Zakov, and Michal Ziv-Ukelson. Sparse RNA Folding: Time and Space Efficient Algorithms. In *Annual Symposium on Combinatorial Pattern Matching*, pages 249–262. Springer, 2009.

5 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). STOC, 2015.

6 Arturs Backurs and Krzysztof Onak. Fast algorithms for parsing sequences of parentheses with few errors. In *PODS*, 2016.

7 Ulrike Brandt and Ghislain Delepine. Weight-reducing grammars and ultralinear languages. *RAIRO-Theoretical Informatics and Applications*, 38(1):19–25, 2004.

8 Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia V. Williams. Truly subcubic algorithms for language edit distance and RNA folding via fast bounded-difference min-plus product. FOCS, 2016.

9 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. FOCS, 2015.

10 J. A. Brzozowski. Regular-like expressions for some irregular languages. IEEE Annual Symposium on Switching and Automata Theory, 1968.

11 Noam Chomsky. On certain formal properties of grammars. *Information and control*, 2(2).

12 Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13, 1970.

13 Sheila A Greibach. The unsolvability of the recognition of linear context-free languages. *Journal of the ACM (JACM)*, 13(4):582–587, 1966.

14 Steven Grijzenhout and Maarten Marx. The quality of the XML web. *Web Semant.*, 19, 2013.

15 R.R Gutell, J.J. Cannone, Z Shang, Y Du, and M.J Serra. A story: unpaired adenosine bases in ribosomal RNAs. Journal of Mol Biology, 2010.

16 John E Hopcroft and Jeffrey D Ullman. Formal languages and their relation to automata. 1969.

17 O.H. Ibarra and T. Jiang. On one-way cellular arrays,. *SIAM J. Comput.*, 16, 1987.

18 Russell Impagliazzo and Ramamohan Paturi. Complexity of k-sat. CCC, pages 237–240, 1999.

19 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? FOCS, pages 653–662, 1998.

20 Mark Johnson. PCFGs, Topic Models, Adaptor Grammars and Learning Topical Collocations and the Structure of Proper Names. ACL, 2010.

21 Ik-Soon Kim and Kwang-Moo Choe. Error repair with validation in LR-based parsing. *ACM Trans. Program. Lang. Syst.*, 23(4), July 2001.

22 Donald E Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

23 Flip Korn, Barna Saha, Divesh Srivastava, and Shanshan Ying. On repairing structural problems in semi-structured data. VLDB, 2013.

24 Martin Kutriba and Andreas Malcher. Finite turns and the regular closure of linear context-free languages. *Discrete Applied Mathematics*, 155(5), October 2007.

25 Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, (49), 2002.

26 Andreas Malcher and Giovanni Pighizzini. Descriptional complexity of bounded context-free languages. *Information and Computation*, 227, 2013.

27 Christopher D Manning. *Foundations of statistical natural language processing*, volume 999. MIT Press.

28 Darnell Moore and Irfan Essa. Recognizing multitasked activities from video using stochastic context-free grammar. NCAI, 2002.

29 E. Moriya and T. Tada. On the space complexity of turn bounded pushdown automata. *Internat. J. Comput*, (80), 2003.

30 Gene Myers. Approximately matching context-free languages. *Information Processing Letters*, 54, 1995.

31 Geoffrey K Pullum and Gerald Gazdar. Natural languages and context-free languages. *Linguistics and Philosophy*, 4(4), 1982.

32 Sanguthevar Rajasekaran and Marius Nicolae. An error correcting parser for context free grammars that takes less than cubic time. *Manuscript*, 2014.

33 Andrea Rosani, Nicola Conci, and Francesco G. De Natale. Human behavior recognition using a context-free grammar.

34 Barna Saha. The Dyck language edit distance problem in near-linear time. FOCS, pages 611–620, 2014.

35 Barna Saha. Language edit distance and maximum likelihood parsing of stochastic grammars: Faster algorithms and connection to fundamental graph problems. FOCS, pages 118–135, 2015.

36 Jose M Sempere and Pedro Garcia. A characterization of even linear languages and its application to the learning problem. In *International Colloquium on Grammatical Inference*, pages 38–44. Springer, 1994.

37 Jeffrey Ullman and John Hopcroft. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001.

38 Leslie G Valiant. General context-free recognition in less than cubic time. *Journal of computer and system sciences*, 10(2), 1975.

39 Balaji Venkatachalam, Dan Gusfield, and Yelena Frid. Faster Algorithms for RNA-Folding Using the four-Russians Method. WABI, 2013.

40 Robert A. Wagner. Order-n correction for regular languages. *Communications of the ACM*, 17(5), 1974.

41 Ye-Yi Wang, Milind Mahajan, and Xuedong Huang. A unified context-free grammar and n-gram model for spoken language processing. ICASP, 2000.

42 Glynn Winskel. *The formal semantics of programming languages: an introduction.* 1993.

43 D.A. Workman. Turn-bounded grammars and their relation to ultralinear languages. *Inform. and Control*, (32), 1976.

44 Shay Zakov, Dekel Tsur, and Michal Ziv-Ukelson. Reducing the worst case running times of a family of RNA and CFG problems, using Valiant's approach. In *WABI*, 2010.