Language Edit Distance Approximation via Amnesic Dynamic Programming

Rajesh Jayaram* Barna Saha†

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, a remarkable result of Abboud, Backers and V. Williams (FOCS 2015) showed that this is essentially optimal; moreover, a combinatorial $o(n^3)$ algorithm is unlikely to exist for the general parsing problem. Language edit distance is a significant generalization of the parsing problem, which computes the minimum edit distance of a given string (insertion, deletion and substitution) to any valid string in the language. 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. However, 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 propose a generic technique of *amnesic dynamic programming* which, given any high-dimensional dynamic programming problem, selectively forgets some of the intermediate states. By performing fewer look-ups, this speeds up the running time at the cost of returning an approximate answer. We believe that this technique will find widespread applications.

Our main result is an $O(n^2)$-time additive approximation algorithm for computing language edit distance to an important subclass of context free grammars, known as *ultralinear* grammars, using this method of amnesic dynamic programming. Starting with the regular languages, which can be parsed in linear time, one well-known hierarchy of languages in increasing order of expressiveness is: linear languages (quadratic parsing time), metalinear and superlinear languages (quadratic parsing time), ultralinear languages (current parsing time $O(n^\omega)$/ $O(n^3)$ fast matrix multiplication/combinatorial), and finally the context free languages. Interestingly, we show that the same conditional lower bound for parsing context free grammars holds for the class of ultralinear grammars as well – this clearly exhibits the demarcation where parsing becomes hard. Therefore, no multiplicative approximation in $o(n^\omega)$ time is possible for the ultralinear language edit distance problem. Finally, we derive quadratic-time language edit distance algorithms for the linear, metalinear and superlinear languages, and show matching conditional lower bounds.

*Brown University, rajesh_jayaram@brown.edu. This work is done as part of the NSF-funded REU program at the University of Massachusetts Amherst.

†University of Massachusetts Amherst, barna@cs.umass.edu.This work is partially supported by a NSF CCF 1464310 grant, a Yahoo ACE Award and a Google Faculty Research Award.

1 Introduction

Developing fast algorithms for dynamic programming (DP) is of paramount interest due to the universality of DP-based algorithms and their frequent high time complexity. Many techniques such as the *Four-Russians method* have been developed in the literature towards this goal [21, 1, 49]. Unfortunately, their speed-up gains have mostly been restricted to only poly-logarithmic factors. There are many applications, however, where one can tolerate some suboptimality. Thus, if we allow for an approximate answer, a major improvement in running time may be possible. In this paper, we propose one such technique of *amnesic dynamic programming* to achieve precisely this.

The technique of DP is motivated by the concept of memoization. It implements a recursive procedure, but stores the results from each computed subproblem in a table which can be accessed multiple times, often leading to a dramatic improvement from exponential to polynomial running time. Still maintaining and accessing the entire DP table can be costly both in terms of time and space. High-degree of polynomial time and space complexity are therefore common drawbacks of a DP algorithm.

The idea of amnesic dynamic programming is simple: instead of maintaining the entire DP table, we selectively forget states of the dynamic programming, therefore computing only a partial table. When computing a solution for a subproblem, there are therefore less look-ups to do. Such a notion of amnesic DP has previously been used [46, 17], however, with the motivation of saving space. For example, Saks and Seshadri obtained an elegant amnesic DP algorithm for computing distance to monotonicity reducing the space requirement from $O(n)$ to $O(\log n)$, n being the input size [46]. While the best DP algorithm for distance to monotonicity has a running time of $O(n \log n)$, their algorithm runs in time $O(n(\log n)^2)$. Nonetheless, we believe that *amnesic dynamic programming* can be instrumental in developing fast approximation for problems with higher-dimensional DPs. In this paper, we use the fundamental problem of *language edit distance* to showcase the strength of this technique. We design a simple deterministic amnesic DP, and provide a framework for analysis. Our framework can be utilized to analyze more complex randomized strategies of forgetting DP states, and can find many more applications.

Language Edit Distance, Hardness of Parsing & an n^ω barrier. Introduced by Chomsky in 1956 [Cho59], context-free grammars (CFG) play a fundamental role in the development of formal language theory [2, 29], compiler optimization [22, 53], natural language processing [35, 40], with diverse applications in areas such as computational biology [43, 9], machine learning [26, 37, 5] and databases [31, 19, 44]. 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, [15] etc. are based on a natural dynamic programming, and run in $O(n^3)$ time[1]. In 1975, in a theoretical breakthrough, Valiant proved a reduction from parsing to Boolean matrix multiplication: the parsing problem can be solved in $O(n^\omega)$ time [48]. Despite decades of efforts, these running times have remain literally unchanged.

Nearly three decades after Valiant's result, Lee came up with an ingenious reduction from the Boolean matrix multiplication to CFG parsing, that for the first time indicated why known parsing algorithms may be optimal [33]. A remarkable recent result of Abboud, Backurs and V. Williams made her claims concrete [4]. Basing on a conjecture of hardness of computing large cliques in graphs, they ruled out any improvement beyond Valiant's algorithm; moreover there can be no combinatorial algorithm for CFG parsing that runs in truly subcubic $O(n^{3-\epsilon})$ time for $\epsilon > 0$ [4]. 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

[1] dependency on the grammar size if not specified is either $|G|$ as in most combinatorial algorithms, or $|G|^2$ as in most algebraic algorithms.

has sought to discover subclasses of context free grammars that are sufficiently expressive yet admit efficient parsing time [34, 32, 23]. Unfortunately, there still exist important subclasses of CFG for which neither better parsing algorithms are known, nor there exist conditional lower bounds to rule out the possibilities.

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 (Language Edit Distance (LED))**.** Given a formal language $\mathcal{L}(G)$ generated by a grammar G over alphabet Σ, and a string $x \in \Sigma^*$, compute the minimum number of edits (insertion, deletion and substitution) needed on 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, 39, 44, 45, 12, 4, 41, 8, 28]. It also 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 $O(n^3)$ time [3], which was improved to $O(|G|n^3)$ by Myers in 1985 [39]. 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 [12]. When considering approximate answers, a *multiplicative* $(1+\epsilon)$-approximation for LED has been presented in [45], 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 [33, 4], LED cannot be approximated within 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 $\epsilon > 0$ [4]. LED provides a very generic framework for modeling problems with vast applications [31, 27, 51, 36, 42, 40, 20]. A fast exact or approximate algorithm for it is likely to have tangible impact, yet there seems to be a bottleneck to improve its running time beyond $O(n^\omega)$ or design a truly subcubic combinatorial algorithm even with approximation. Can we break this n^ω barrier?

Additive Approximation & Hierarchy within CFG. One possible approach to break this barrier is to allow *additive approximation.* Since the hardness of multiplicative approximation arise from the lower bound of parsing, it is possible to go below n^ω and design a purely combinatorial algorithm for LED with 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 [45, 12], this may imply new algorithms for many other fundamental problems. In this paper, we make a significant progress in this direction. While, we are not yet able to tackle the entire class of CFGs, we consider an important subclass of CFG, known as the *ultralinear grammar* [55, 14, 34, 11, 38] and show (1) an $\Omega(n^\omega)$ hardness result for parsing ultralinear grammars, and (2) an additive approximation for language edit distance to ultralinear grammar than runs in $O(|G|n^2)$ time using amnesic dynamic programming.

Let us use $G = (Q, \Sigma, P, S)$ to denote a grammar where Q is the set of nonterminals, Σ is the alphabet or terminals, P is the set of productions, and S is a special nonterminal designated as the start state.

Definition (ultralinear)**.** A grammar $G = (Q, \Sigma, P, S)$ is said to be **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 $A \in Q_i$ and $\alpha \in \Sigma$, or *non-linear productions* of the form

$$X \to w, \text{ where } w \in (\Sigma \cup Q_1 \cup Q_2 \cup \cdots \cup Q_{i-1})^*$$

If we allow k to be ∞ then we get the entire class of context free grammars. In fact, for any k, we can restrict any context free grammar G to a k-ultralinear grammar G', $\mathcal{L}(G') \subseteq \mathcal{L}(G)$, and the results of our paper holds for such a $\mathcal{L}(G')$. It is precisely this procedure of creating a k-ultralinear grammar G' from a CFG G that we use in our proof of hardness for parsing ultralinear languages (Theorem 13). For example, if G is the well-known *Dyck Languages* [44, 8], the language of well-balanced parenthesis, $\mathcal{L}(G')$ contains the set of all parentheses strings with at most k-levels of nesting. As an another example, consider RNA-folding [12, 49, 56] which is a basic problem in computational biology and can be modeled by grammars. $\mathcal{L}(G')$ for RNA-folding denotes the set of all RNA strings with at most k-nested folds. In typical applications, we do not expect k to be too large [19, 31, 6].

The ultralinear languages are precisely the class of languages that are accepted by a finite-turn pushdown automata [16], forming a powerful link between the theory of formal languages and automata theory. They are also known as the *non-terminal bounded* grammars. Moreover, they are a part of an important hierarchy shown in Figure 1. Starting with the regular languages, which can be parsed in linear time, the hierarchy successively moves to more expressive grammars: linear languages (see Section 2), metalinear and superlinear languages (see Section 6), ultralinear languages, and finally to the context free languages. It is known that ultralinear languages strictly contain the metalinear languages [10], metalinear languages generalize linear languages which in turn contain regular languages.

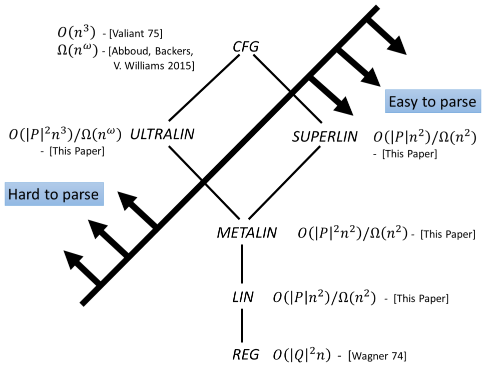


Figure 1: CFG Hierarchy & Our Results

It is known that the edit distance to regular languages can be computed in $O(|G|^2n)$ time [50], however no such fast algorithms have been developed for computing the edit distance to any of these more expressive languages. Among our other results we develop language edit distance algorithms to linear, metalinear, and superlinear languages that run in time quadratic in n. Moreover, we show matching lower bound assuming the Strong Exponential Time Hypothesis [24, 25].

Interestingly, till date there exists no parsing algorithm for the ultralinear grammars that run in time $o(n^\omega)$, while $O(n^2)$ 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!

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 [13, 7], to linear language edit distance. The grammar in our construction has constant size, and thus demonstrates a tight reduction.

Theorem (12)**.** *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 [4], 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. We then demonstrate the same hardness result for our grammar, showing that for any ultralinear grammar G it is unlikely that any $o(\text{poly}(|G|)n^c)$ algorithm exists for the ultralinear language recognition problem, where $c = 3$ for combinatorial algorithms and $c = \omega$ for any algorithm. Precisely, we prove the following theorem.

Theorem (13)**.** *There is a ultralinear grammar $\mathcal{G}_U^\ell = \mathcal{G}_U$ such that if we can solve the membership problem for string of length n in time $O(|\mathcal{G}_U|^\alpha n^c)$, where $\alpha > 0$ is some fixed constant, then we can solve the k-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 5), superlinear (Theorem 15), and metalinear language edit distance (Theorem 16), 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 [4]. 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, which utilizes the technique of *amnesic dynamic programming.* 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, into a family of amnesic dynamic programming algorithms which produce *additive approximations* of the edit distance, and provide a tool for proving general bounds on any such algorithm. We believe that our amnesic technique can be applied to wide range of potential dynamic programming approximation algorithms, and lends itself particularly well to randomization. We then present two explicit schemes for forgetting dynamic programming states. In particular, for any k-ultralinear grammar, after fixing a sensitivity constant γ, we demonstrate an $O(2^k\gamma)$ additive approximation in $O(|P|(n^2 + (\frac{n}{\gamma})^3))$ time, and an $O(k\gamma)$ additive approximation in $O(|P|(n^2 + \frac{n^3}{\gamma^2}))$ time.

Theorem 1. *If $\mathcal{A}$ is a γ-uniform grid approximation, then the edit distance computed by $\mathcal{A}$ satisfies*

$$|OPT| \leq |\mathcal{A}| \leq |OPT| + O(2^k\gamma)$$

and it runs in in $O(|P|(n^2 + (\frac{n}{\gamma})^3))$ time.

Theorem 2. *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.

2 Linear Grammar Edit Distance in Quadratic Time

Definition (linear grammar)**.** A grammar $G = (Q, \Sigma, P, S)$ is said to be **linear** if every production is of the form $A \to \alpha B$ or, $A \to B\alpha$, or $A \to \alpha$ for $A, B \in Q$, and $\alpha \in \Sigma$. Such productions are referred to as *linear* productions.

Note that if we can 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. 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[18, 47].

Given a linear grammar $G = (Q, \Sigma, P, S)$, and a string $\overline{x} = x_1x_2 \ldots x_n \in \Sigma^*$, we give an $O(n^2|P|)$ algorithm to compute edit distance between $\overline{x}$ and G in this section. The primary motivation for doing this will be to develop a construction that will lie at the heart of our main approximation algorithm in Section 3.

Definition. For any nonterminal $A \in Q$, define $null(A)$ to be the length of the shortest string in Σ^* derivable from A.

For the algorithms in this paper, it will be necessary to pre-compute the values of $null(A)$. This occurs in the case where we would like to compute the cost of deriving ε from A, since the minimum edit distance between ε and the set of strings derivable from A is precisely $null(A)$. Since this cost consists of deleting a whole string produced by A, $null(A)$ can be thought of as a set of deletion errors. Computing this can be done in multiple ways, and the running time is independent of n. The following theorem will allow us to pre-compute and store the value of $null(A)$ for all $A \in Q$, before running the language edit distance algorithms, thus we assume for the rest of the paper that this has already been done.

Theorem 3. *The set of values* $\{null(A) \mid A \in Q\}$ *can be computed in* $O(|Q||P|\log(|Q|))$ *time.*

Algorithm. Fundamental to our algorithm is the construction of a weighted digraph $\mathcal{T} = \mathcal{T}(G, \overline{x})$ from G and $\overline{x}$ 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. When the grammar and input string are fixed, we omit the arguments and simply write $\mathcal{T}$.

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 clouds, $1 \le i \le j \le n$, corresponding to the substring $x_ix_{i+1}....x_j$. 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 (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. The edges of $\mathcal{T}$ will correspond to the productions in G. Thus the vertex set of $\mathcal{T}$ is $V(\mathcal{T}) = \{A_k^{i,j} \mid 1 \le i \le j \le n,\ 1 \le k \le q\} \cup \{t\}$. 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 $x_ix_{i+1} \ldots x_j$. 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.

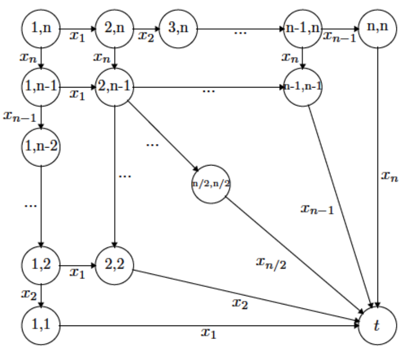


Figure 2: Clouds corresponding to Linear Grammar Edit Distance Graph Construction. Each cloud contains a vertex for every nonterminal

Adding the edges. Each edge in $\mathcal{T}$ is directed, has a weight in $\mathbb{Z}^+$ and a label from $\{x_1, x_2, .., x_n, \epsilon\} \cup \{\epsilon(\alpha) \mid \alpha \in \Sigma\}$. 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. Given input $x_1x_2 \ldots x_n$, for all nonterminals A_k, A_t and every $1 \le i \le j \le n$, the construction is as follows:

- **Legal Productions:** For $i \ne j$, then if $A_k \to x_iA_t$ is a production, add the edge $A_k^{i,j} \xrightarrow[0]{x_i} A_t^{i+1,j}$ to $\mathcal{T}$. If $A_k \to A_tx_j$ is a production, add the edge $A_k^{i,j} \xrightarrow[0]{x_j} A_t^{i,j-1}$ to $\mathcal{T}$.

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

The goal is then to find the shortest path from $S = A_1^{1,n} \to t$. We now prove that the weight of this shortest path is the minimum language edit distance from $\overline{x}$ to G.

Remark 1. *This approach based on shortest path computation is slightly different from that taken by Aho and Peterson [3], which uses the construction of a* covering grammar*. We can also follow [3] and add error-producing rules corresponding to insertion, deletion and substitution errors. These do not keep the grammar linear any more. However, we can still modify the algorithm of [3] somewhat, and get the desired running time. In order to find the edit distance between substring $x_i x_{i+1} \ldots . x_j$ and G, it is enough to compute the distance of x_i, and $x_{i+1} x_{i+2} \ldots .. x_j$ to G, and the edit distance of $x_i x_{i+1} \ldots x_{j-1}$ and x_j to G, and take the minimum of the two. Since, time spent on each substring is constant, and there are $O(n^2)$ substrings, this results in an $O(|P|n^2)$ algorithm. This alternate approach taken here is better suited to the specific graph constructions utilized in this paper for more involved approximation algorithms to follow.*

Theorem 4. *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 $x_i \ldots x_j$ and the set of strings which can be derived from A_k.*

Theorem 5. *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.*

Proof. The cost of the shortest path follows immediately from the previous theorem. Now there are $O(n^2)$ vertices in the graph for every nonterminal $A \in Q$. Hence, there are a total of $O(|Q|n^2)$ vertices. Let P_k denote the set of productions involving A_k on the left hand side. Then, for each $A_k^{i,j}$, the total out degree of that node is $O(|P_k|)$. Hence the total number of edges emanating from cloud (i,j) is $O(|P|)$, resulting in a total of $O(|P|n^2)$ edges. Since the maximum edge weight is bounded by 1, utilizing the best known single-source shortest path algorithm gives a $O(E(T) + V(T)) = O(|P|n^2)^2$ runtime algorithm to compute the weight of the shortest path from $S^{1,n}$ to t, which is the minimum edit distance from $\overline{x}$ to t. □

3 Context Free Language Edit Distance

In this section, we develop an algorithm which utilizes the graph construction presented in Section 2 to compute the language edit distance of a string $\overline{x} = x_1 \ldots x_n$ to any context free grammar $G = (Q, \Sigma, P, S)$. Our algorithm has a cubic running time in n, and thus is not itself an improvement on that of Aho and Peterson [3]. However, by building off of our construction, we will be able to provide *additive approximations* of the edit distance in the next section that

[2] We assume $|P| \geq |Q|$, that is each nonterminal is involved in at least one production on the left.

runs in quadratic time. The bounds given by our approximation algorithms will be particularly useful if G is an ultralinear grammar.

Let us first recall some notations that will be useful in the following sections. Given a context-free grammar $G = (Q, \Sigma, P, S)$, we first introduce a normal form which we will assume for the rest of the paper. The normal form is same as the *Chomsky Normal Form*, except that we allow for the addition of linear productions. We show in Lemma 2 in Appendix that any k-ultralinear grammar can be converted into a k^*-ultralinear language in this normal form, where $k^* \leq k \log(p)$, and p is the maximum number of nonterminals on the right hand side of any production.

Definition. A context-free grammar G is in *normal form* if for any nonterminal $A \in Q$, all productions with A on the left hand side are of the form: (1) $A \to \beta B \mid B\beta \mid \beta$, or (2) $A \to CD$ where $\beta \in \Sigma$ is any terminal, and $B, C, D \in Q$. Productions of the first type are called *linear productions*, and productions of the second type are called *non-linear productions*.

Note that as a result, any word produced by a k-ultralinear grammar in this normal form must be derived using no more than 2^k non-linear productions of type (2). Note that if we consider only the linear productions of the form given by (1), we can construct the linear grammar edit distance graph $\mathcal{T}$ using the same procedure as in Section 2.

Sketch of the Algorithm: Our algorithm makes crucial use of our earlier construction of the linear language edit distance graph $\mathcal{T}$. The essential idea is that, when tasked with deriving some substring $x_i \dots x_j$ from a nonterminal $A \in Q$ using a sequence of productions in P, and error productions corresponding to edit edges, there are two possibilities for the first production. Either the first production is a linear production, creating x_i or x_j with cost 0 if it is a legal production and cost 1 if it is an error production, or it is a non-linear production of the form $A \to CD$. In the later case, no terminal is produced and we are now tasked with deriving $x(i:j) = x_i \dots x_j$ from CD. The first case is handled by the original construction of the graph $\mathcal{T}$. In the second case, C must derive some substring $x(i:\ell)$ and D must derive $x(\ell+1:j)$, each of which is a substring of size less than or equal to that of $x(i:j)$ (equal in the case that either one of C or D must derive all of $x(i:j)$, and the other derives no terminal). To handle this situation, our algorithm computes shortest path on $\mathcal{T}$ in phases, where in each phase we compute shortest path to all substrings of a certain length, so that when computing the cost of the above non-linear production, we will have already computed the minimum cost of deriving $x(i:\ell)$ from C and $x(\ell+1:j)$ from D over all $i \leq \ell \leq j-1$.

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 $\mathcal{T}(G_L, \overline{x}) = \mathcal{T}$, as constructed in Section 2. Let L_i be the set of clouds in $\mathcal{T}$ which correspond to substrings of length i. In other words:

$$L_i = \{(k, j) \in \mathcal{T} \mid j - k + 1 = i\}$$

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. We call this the edge-reversed linear edit distance graph. The idea then is to compute the single source shortest path from the sink to every vertex of $\mathcal{T}^R$ in n phases, where each phase corresponds to substrings of $\overline{x}$ of different length. Our algorithm will maintain the property that 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.

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 to L_i^R, and from t to L_i^R, for all $1 \leq i \leq n$, resulting in an augmented subgraph which we denote $\overline{L}_i^R$. We then compute single source shortest path from t on $\overline{L}_i^R \cup \{t\}$. 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.

   Induction Hypothesis. 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 $A^{p,q}$.

   Having computed shortest paths from t to every vertex in the subgraphs $\overline{L}_1^R, \ldots, \overline{L}_{i-1}^R$, 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 simply the linear production edges created in the linear grammar edit distance algorithm.

4. Now, 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

$$c = \min_{p \le \ell < q} T_{p,\ell}(B) + T_{\ell+1,q}(C)$$

   Additionally, 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. Note again that the direction of the edges are reversed because we are adding edges to L_i^R and not L_i.

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 6. *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.*

Runtime. For every cloud in the graph, we add at most $O(|P|)$ new edges originating at vertices in that cloud. Thus the total number of edges which we run a shortest path algorithm on in steps 2 and 6 is still $O(|P|n^2)$, the same order as the size of the original L_i^R. Thus the total time required to run shortest path in steps 2 and 6 for all $i = 1, \ldots, n$ is $O(|P|n^2)$. Now for each of the $O(n^2)$ clouds in $\mathcal{T}$, running steps 1, 3 and 5 takes $O(|P|)$ time since we add $O(|P|)$ new edges in each cloud, each in constant time. In step 4, for each production in $|P|$ and each of the $O(n^2)$ clouds, the algorithm takes a minimum over at most n values to compute the cost c, thus this step takes $O(|P|n^3)$ time. Thus the total runtime is $O(|P|n^2 + |P|n^3) = O(|P|n^3)$.

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, 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. In particular, for any k-ultralinear language, we give explicit $O(k\sqrt{n})$ and $O(2^k n^{1/3})$ additive approximations from this family which run in quadratic time. We expect that using the general bounds for specific grammars where more is known about the structure of the derivations will allow for even better approximations.

Furthermore, as shown in our construction in the proof of hardness of parsing ultralinear grammars (Section 5), 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 have a derivation tree of height $\leq k$. Thus our technique gives a general method to construct and prove bounds on the language edit distance to any such set $\mathcal{L}(G') \subset \mathcal{L}(G)$.

Family of Approximation Algorithms. We now introduce a family $\mathcal{F}$ of approximation algorithms which utilizes the structure of our exact algorithm from Section 3 .

Definition. 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 $\{(p, q) \mid 1 \leq p \leq q \leq n\}$.
2. **Subset of splitting points.** For every cloud that $\mathcal{A}$ computes non-linear production edge for in step 4, when computing the edge weight c it takes minimum over only a subset of all possible splitting points.

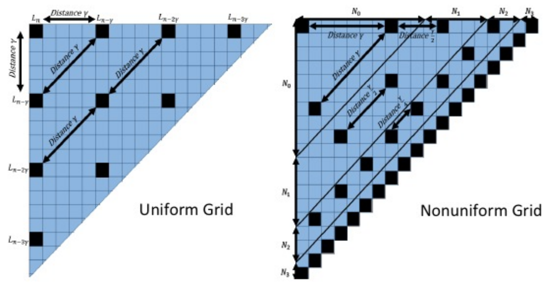


Figure 3: Non-uniform edges are computed only for a subset of the clouds (colored black). Moreover only a subset of the splitting points are considered while computing the cost.

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. Note that since we do not remove any of the $O(|P|n^2)$ vertices of $\mathcal{T}$, running shortest path in steps 2 and 6 of the algorithm still takes $O(|P|n^2)$ time, thus this is a lower bound on the runtime of our approximation algorithm no matter how many edges we forget to construct. 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 1 and 2. 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 $\mathcal{A}$.

Example 1. *An approximation algorithm $\mathcal{A} \in \mathcal{F}$ is a γ-uniform grid approximation if for all $i = n, (n-\gamma), (n-2\gamma), \ldots, (n - \lfloor \frac{n}{\gamma} \rfloor \gamma)$ (see Figure 3 (left))*

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 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 1).

Now for $i = 0, 1, \ldots, \log(n)$, set $N_i = \{L_j \mid \frac{n}{2^{i+1}} \leq j \leq \frac{n}{2^i}\}$. Let $N_i' \subset N_i$ be an evenly-spaced $\frac{2^i}{\gamma}$ fraction of the L_j's in N_i. Then:

Example 2. *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 for a $\frac{2^i}{\gamma}$ evenly-spaced fraction of the clouds in L_j. Furthermore, in any cloud for which $\mathcal{A}$ computes non-linear production edges, $\mathcal{A}$ considers only an evenly-spaced $\frac{2^i}{\gamma}$ 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 for any k-ultralinear grammar G 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 2).

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 $E(\mathcal{T}) \subset E(\mathcal{T}^{\mathcal{A}}) \subset E(\mathcal{T}^{OPT})$.

High-level Steps of the Analysis The analysis follows the following steps.

1. We define the notion of a *binary production-edit tree* which is a binary tree representing the execution of the optimum algorithm, and similarly, a binary production-edit tree to represent the execution of our approximation algorithm $\mathcal{A} \in \mathcal{F}$. Each internal node in the tree is denoted by $[A^{p,q}, B^{r,s}, c]$ and represents a path in $\mathcal{T}$ of only linear productions from $A^{p,q}$ to $B^{r,s}$ with a total cost of c. If the algorithm takes a non-linear edge $B \to CD$, with splitting point $l \in [r, s)$, then two children of $[A^{p,q}, B^{r,s}, c]$ are created that start at $C^{r,l}$ and $D^{l+1,s}$ respectively. The leaf nodes represent path to the sink – for instance $[A^{p,q}, t, c]$, that starts at $A^{p,q}$, follows only linear edges in $\mathcal{T}$ and incurs a cost of c. Using such trees, one can concisely represent the entire execution of both the optimal, and the approximation algorithm.

2. For an approximation algorithm $\mathcal{A} \in \mathcal{F}$, we define precision functions $\alpha(p, q)$ which gives an upper bound on the minimum ℓ_1 distance from the cloud (p, q) to a cloud (r, s), $p \leq r \leq s \leq q$, for which $\mathcal{A}$ constructs non-linear edges, and $\beta(p, q)$ gives an upper bound on the maximum distance between the splitting points considered by $\mathcal{A}$ for constructing the non-linear edges at the cloud (r, s).

3. Finally, we define a *production-edit tree mapping*, which will map the production-edit tree of the optimal algorithm, $\mathbb{T}_{OPT}$, to a production edit tree of our approximation algorithm, $\mathbb{T}$, which we will have to construct. Using the mapping, we will bound the cost of $\mathbb{T}$ in

terms of the cost of the optimal tree $\mathbb{T}_{OPT}$ and the precision functions $\alpha(p,q), \beta(p,q)$. Constructing the tree $\mathbb{T}$ and bounding its cost is a three part process. In Lemma 1, we show how to construct a single node of $\mathbb{T}$ given a node of $\mathbb{T}_{OPT}$. Then in Theorem 8, we repeatedly use Lemma 1 to construct the entire tree $\mathbb{T}$ from the top down. During this process, we accumulate a set of inequalities which bound the costs of the nodes we construct. Finally, in Theorem 9 we sum over this set of inequalities to obtain a bound on the entire cost of $\mathbb{T}$, from which our final theorem, Theorem 10, follows.

Binary Production-Edit Tree. We now introduce the main structure which we will use to compare the solution produced by the exact algorithm to the solution produced by an $\mathcal{A} \in \mathcal{F}$. The intuition is as follows. The normal form we have imposed on our grammar G partitions the set of productions into linear productions and non-linear productions with exactly two nonterminals on the right hand side, so in any derivation we need only consider each of these cases. Since all error productions are linear, the process of deriving an input string $\overline{x}$ from our grammar, possibly using error productions along the way, can be represented through a certain binary tree $\mathbb{T}$, where each node stores a sequence of linear productions. These sequences of linear productions are just sequences of edges created in step 3 of the exact algorithm (the linear edges), or equivalently a path in the linear grammar edit distance graph $\mathcal{T}$. The root of $\mathbb{T}$ consists of a sequence of linear productions made starting from S in some derivation of $\overline{x}$. Once the first non-linear production is made, $A \to BC$, the sequence of linear productions stored in the root terminates, and left and right children of the root of $\mathbb{T}$ are created, starting at the nonterminals B and C respectively. Then starting from each of B and C, a new sequence of linear productions is stored in the corresponding nodes of $\mathbb{T}$, each terminating when the first non-linear production is made, and so on. We formalize this in the following definition.

Definition. A *Binary Production-Edit Tree* $\mathbb{T}$ for G and $\overline{x}$ is a binary tree which satisfies the following properties:

1. Each node of $\mathbb{T}$ stores a path in the linear graph $\mathcal{T} = \mathcal{T}(G, \overline{x})$. 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 of $\mathcal{T}$, then v must have two children, v_r, v_l, such that there exists a production $B \to CD$ for some nonterminals $C, D \in Q$ so that the starting vertices of the paths given by v_l and v_r are $C^{r,\ell}$ and $D^{\ell+1,s}$ respectively, where ℓ is some splitting point $r - 1 \le \ell \le s$. 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 with a cost of either $null(C)$ or $null(D)$, indicating that one of either C or D will be nullified.

3. Now if $v \in \mathbb{T}$ is any node such that the corresponding path ends at the sink t of $\mathcal{T}$, then v is a leaf of $\mathbb{T}$. Conversely, a node v is a leaf of $\mathbb{T}$ if and only if it is either a nullified node, or it corresponds to such a path ending at the sink of $\mathcal{T}$.

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 for which our algorithms compute shortest path (such as $\mathcal{T}, \mathcal{T}^{\mathcal{A}}, \mathcal{T}^{OPT}, \mathcal{T}_{NL}$, ect), we will use the term *vertex*. When speaking about any production-edit tree $\mathbb{T}$, we will use the term *node* to describe the elements of the vertex set of $\mathbb{T}$.

We can completely describe the output of the exact algorithm or of any $\mathcal{A} \in \mathcal{F}$ by such a tree $\mathbb{T}$. If at any point, one of the non-linear edges constructed in steps 4 or 5 is used along a path taken in the algorithm, then the node in $\mathbb{T}$ corresponding to that path terminates and two children are created. Thus each path given by a node in $\mathbb{T}$ will terminate either when a non-linear production $A \to BC$ is made, or when the sink is reached via linear edges alone. A node of the first type will have two children, a right child given by a path in $\mathcal{T}$ beginning at $B \in Q$, and the left by a path in $\mathcal{T}$ beginning at $C \in Q$. A node of the second type corresponds to a leaf in the tree.

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. We use this notation since we will only need to consider the starting and ending vertices of such a path and its resulting cost. 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.

We can now represent any sequence of edits produced by a language edit distance algorithm by such a production-edit tree, where the edit distance is given by the sum of the costs stored in the nodes of the tree. In particular, we have a tree $\mathbb{T}_{OPT}$ given by a solution produced by an optimal algorithm. For any production-edit tree $\mathbb{T}$, we denote the associated total cost by $\|\mathbb{T}\|$. To be precise, if $[\cdot, \cdot, c_1], \dots, [\cdot, \cdot, c_k]$ is the set of all nodes in $\mathbb{T}$, then $\|\mathbb{T}\| = \sum_{i=1}^{k} c_i$. Notice that given a fixed approximation algorithm $\mathcal{A} \in \mathcal{F}$, only certain production-edit trees $\mathbb{T}$ could actually correspond to the solution that $\mathcal{A}$ produces, giving rise to a natural definition.

Definition (Production-Edit Trees for $\mathcal{A}$)**.** For an approximation algorithm $\mathcal{A} \in \mathcal{F}$, let $\mathcal{D}_{\mathcal{A}}$ be the set of production-edit trees $\mathbb{T}$ which satisfy the following constraints:

1. If $[A^{p,q}, B^{r,s}, \cdot]$ is a node in a $\mathbb{T}$, 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 compute the splitting point ℓ for 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 production-edit trees 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 $\mathcal{T}^{\mathcal{A}}$. Now we would like for this shortest path to corresponds to an optimal $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$. In other words, we would like to show that $\mathcal{A}$ returns the production-edit tree $\mathbb{T}_{\mathcal{A}}$ such that $\mathbb{T}_{\mathcal{A}} = \arg\min_{\mathbb{T} \in \mathcal{T}_{\mathcal{A}}} \|\mathbb{T}\|$. We now do precisely this.

Theorem 7. *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 production edit tree 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 $c \geq \|\mathbb{T}_{OPT}\|$. 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 $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$.

Precision Functions. We now fix such an approximation algorithm $\mathcal{A}$, and define associated precision functions $\alpha(p, q), \beta(p, q)$. These functions will provide an upper bound on how poorly the graph $\mathcal{T}^{\mathcal{A}}$ approximates $\mathcal{T}^{OPT}$ near the cloud (p, q). Specifically, $\alpha(p, q)$ controls the approximation factor due to step 1 (not computing non-linear edges), and $\beta(p, q)$ controls the approximation factor due to step 2 of $\mathcal{A} \in \mathcal{F}$ (not computing splitting points). First, for any two clouds (p, q) and (r, s) with $p \leq r$ and $q \geq s$, define the *distance between clouds* to be $d_c((p, q), (r, s)) = (r - p) + (q - s)$. This is essentially the ℓ_1 norm, with the exception that we require $x(r : s)$ to be a substring of $x(p : q)$.

Definition (Precision Functions)**.** For any cloud $(p,q) \in \mathcal{T}^{\mathcal{A}}$, let $\alpha(p,q)$ be any upper bound on the minimum distance $d_c((p,q),(r,s))$ such that $\mathcal{A}$ 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 $d_c((p,q),(r,s)) \leq \alpha(p,q)$. Furthermore, the precision functions 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 select* $\mathcal{A}$ from some specified distribution over $\mathcal{F}$.

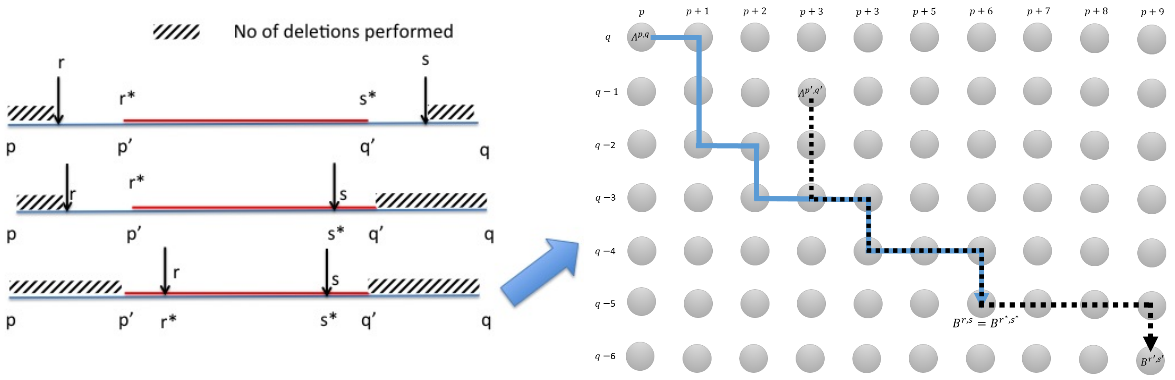
Constructing a tree $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$ similar to $\mathbb{T}_{OPT}$. Our goal will now be to construct a tree $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$ for some $\mathcal{A} \in \mathcal{F}$. We will do this by considering each node v of $\mathbb{T}_{OPT}$, and constructing 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 corresponding to v ends having taken a non-linear production edge in a cloud that $\mathcal{A}$ does not compute non-linear edges for. Every time this happens, we will need to find and move to the closest possible cloud which $\mathcal{A}$ *does* consider before making the same non-linear production that the exact algorithm did. After doing this, the resulting child paths will deviate from those 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 8. The lemma maps a node $[A^{p,q}, B^{r,s}, c] \in \mathbb{T}_{OPT}$, to a node $[A^{p',q'}, B^{r',s'}, c'] \in \mathbb{T}$, given that there is some overlap between the substrings $x(p:q)$ and $x(p':q')$, such that the size of c' is bounded.

Lemma 1. *Let $[A^{p,q}, B^{r,s}, c]$ be any non-leaf node in $\mathbb{T}_{OPT}$, where $A^{p,q}, B^{r,s} \in \mathcal{T}^{OPT}$, and let $\mathcal{A} \in \mathcal{F}$ be an approximation algorithm with precision functions $\alpha(p,q), \beta(p,q)$ and production-edit tree $\mathbb{T}_A = \mathbb{T}$ such that $[A^{p',q'}, B^{r',s'}, c'] \in \mathbb{T}$. 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.*

Proof. Let $e_1 \dots e_\ell$ be the sequences of edges taken by the path corresponding to $[A^{p,q}, B^{r,s}, c] \in \mathbb{T}_{OPT}$. We construct a corresponding sequence of edges $e'_1 \dots e'_\ell$ from $A^{p',q'}$, where e'_j will correspond to the same production as e_j, but with potentially higher cost. In other words, if e_j corresponds to the production $A \to \gamma B$, then e'_j will be an edge corresponding to the same production, however it may be a deletion edge $A \xrightarrow[1]{\epsilon(\gamma)} B$ while e_j is not. We will change a non-deletion edge e_j to a deletion edge if and only if the corresponding production produces a terminal x_ν such that $\nu \notin \{p', p'+1, \dots, q'\}$, meaning it is not in the substring that we need to derive from $A^{p',q'}$. We need to consider several cases based on the overlap between $x(p:q)$ and $x(p':q')$

(1) If $p \leq p'$ and $q \geq q'$, then $x(p':q')$ is a substring of $x(p:q)$. For all $1 \leq j \leq \ell$, if e_j produces a terminal x_ν with $p \leq \nu < p'$ or $q' < \nu \leq q$, set e'_j to be the deletion edge corresponding to the same production. Otherwise set e'_j to be the edge corresponding to the same production as e_j (albeit e'_j may be in a different cloud than e_j). There can be at most $|p'-p| + |q'-q|$ such extra deletion edges, thus $e'_1 \dots e'_\ell$ pay at most $c + |p'-p| + |q'-q|$. In general, there may be fewer extra deletion edges, as the substring corresponding to the ending vertex $B^{r,s}$ of OPT may not be a substring of $x(p':q')$.

Notice that at any point j, after taking the edges $e'_1, \dots, e'_j$ from $A^{p',q'}$, we will be at a vertex labeled with the same nonterminal as after taking $e_1, \dots, e_j$ starting from $A^{p,q}$. Thus, we arrive at a vertex B^{r^*,s^*} after $e'_1, \ldots, e'_l$. Furthermore, at every point the substring considered by the cloud originating from e'_j will be a subset of the string considered by the cloud originating from e_j. Thus after taking $e'_1 \ldots e'_\ell$, when we arrive at B^{r^*,s^*}, we must have $(s^* - r^*) \leq (s - r)$. Now every time we change an edge e_j that was not a deletion edge to a deletion edge e'_j, the path of OPT becomes one cloud closer to the path that we are in. Thus, if we change d (note that $d \leq |p' - p| + |q' - q|$) of the edges into deletion edges, we have that the cloud (r^*, s^*) is distance d closer to the cloud (r, s) than (p', q') was to (p, q). Therefore, the extra deletion cost paid is at most $(|p' - p| + |q' - q|) - (|r^* - r| + |s^* - s|)$. Thus the total cost we pay to reach B^{r^*,s^*} is at most $c + (|p' - p| + |q' - q|) - (|r^* - r| + |s^* - s|)$.



Now $(s^* - r^*) \leq (s - r)$, hence, there exists a cloud (r', s') for which non-linear production edges have been computed, such that $d((r^*, s^*), (r', s')) \leq \alpha(r, s)$. Thus from B^{r^*,s^*} we can take insertion edges $B^{r^*,s^*} \to \ldots \to B^{r',s'}$ arriving at the desired vertex at additional cost at most $\alpha(r, s)$. Thus for non-leaf nodes the total cost is at most $c + (|p' - p| + |q' - q|) - (|r^* - r| + |s^* - s|) + \alpha(r, s)$. Since $d((r^*, s^*), (r', s')) = |r^* - r'| + |s^* - s'| \leq \alpha(r, s)$, by the triangle inequality we have $(|r^* - r| + |s^* - s|) \geq (|r' - r| + |s' - s| - \alpha(r, s))$.

Note that if the node in question is a leaf $[A^{p,q}, t, c]$, then we need not pay this extra cost since after following $e'_1 \ldots e'_\ell$ we will already be at the sink. Thus the total cost is at most $c + (|p' - p| + |q' - q|) - (|r' - r| + |s' - s|) + 2\alpha(r, s)$ for non-leaf nodes as desired, and $c + (|p' - p| + |q' - q|)$ for leaf nodes.

Illustrated in the figure on the left is an example of the path $[A^{p,q}, B^{r,s}, c]$ and the corresponding constructed path $[A^{p',q'}, B^{r',s'}, c']$ when $x(r, s)$ is a substring of $x(p', q')$. $[A^{p,q}, B^{r,s}, c]$ is given by the filled in path, and $[A^{p',q'}, B^{r',s'}, c']$ is the dotted path. Each circle is a cloud, and by construction whenever the paths meet at a cloud they will necessarily be at the same vertex within the cloud.

(2) If $p > p'$ and $q \geq q'$, in this case we need to first follow some insertion edges before we can apply the argument from case **(1)**. We set $l = p - p'$ and create an edge e''_j to be the insertion edge that inserts $x_{p'+j-1}$ for $1 \leq j \leq l$. Following the edges, $e''_1, \ldots e''_l$ we pay a cost of at most $p - p'$ and arrive at a vertex $A^{p,q'}$. Note that for every insertion edge we travel across, the cloud our path is in becomes one cloud closer to the starting cloud of OPT. Now starting from $A^{p,q'}$, we are back in case **(1)** where now the distance between the beginning clouds (p, q') and (p, q) is $|q - q'|$. Thus by the argument from the first case we can reach a vertex $B^{r',s'}$ from $A^{p,q'}$ with cost at most $c + |q' - q| - (|r' - r| + |s' - s|) + 2\alpha(r, s)$ for non-leaf nodes, and cost at most $c + |q' - q|$ for leaf nodes. Since we paid at most $p - p'$ to get to $A^{p,q'}$ from $A^{p',q'}$, the total cost is at most $c + (|p' - p| + |q' - q|) - (|r' - r| + |s' - s|) + 2\alpha(r, s)$ for non-leaf nodes, and $c + (|p' - p| + |q' - q|)$ for leaf nodes as desired.

The case where $p \leq p'$ and $q < q'$ is symmetric, as we simply start by taking edges that insert $x_{q+1}, \ldots, x_{q'}$ instead of $x_{p'}, \ldots, x_{p-1}$. Finally, if both $p > p'$ and $q < q'$, we take edges inserting both $x_{q+1}, \ldots, x_{q'}$ and $x_{p'}, \ldots, x_{p-1}$, paying a cost of $(|p' - p| + |q' - q|)$ along the way, and then we can return to case **(1)** starting at $A^{p,q}$, from which we pay a further cost of at most

$c-(|r'-r|+|s'-s|)+2\alpha(r,s)$ to reach $B^{r',s'}$ for non-leaf nodes, and cost at most c to reach the sink for leaf nodes. Thus in all cases the cost is at most $c+(|p'-p|+|q'-q|)-(|r'-r|+|s'-s|)+2\alpha(r,s)$ and $c+(|r'-r|+|s'-s|)$ for non-leaf and leaf nodes respectively. Note that in all cases, every time OPT took an edge that derived a terminal in $x(p':q')$, our path also took an edge which derived the same terminal. Thus the ending clouds produced in all cases satisfy $r \leq r' \leq s' \leq s$. □

Recall that to construct the explicit tree $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$, we will need to consider each node v of $\mathbb{T}_{OPT}$ and then construct a corresponding node u in $\mathbb{T}$, such that the path stored in u *imitates* the path in v as closely as possible. To precisely define the notion of a node in $\mathbb{T}$ *corresponding* to a node in $\mathbb{T}_{OPT}$, we define a mapping from $\mathbb{T}_{OPT} \to \mathbb{T}$ which will send a node v to its corresponding node u. For the purposes of our argument, such a mapping must satisfy several specific properties. We introduce these now.

Definition. Let $\mathbb{T}_1, \mathbb{T}_2$ be any two production-edit trees, and $V(\mathbb{T}_1), V(\mathbb{T}_2)$ be the corresponding node sets. Then a function $\psi : V(\mathbb{T}_1) \to V(\mathbb{T}_2)$ is a *Production-Edit Tree Mapping* if it is surjective and if

1. ψ maps the root of $\mathbb{T}_1$ to the root of $\mathbb{T}_2$.
2. Every non-leaf node of $\mathbb{T}_2$ is mapped to by at most one node in $\mathbb{T}_1$.
3. If $v_1 \to v_2$ is an edge in $\mathbb{T}_1$, then either $\psi(v_1) \to \psi(v_2)$ is an edge in $\mathbb{T}_2$, or $\psi(v_1) = \psi(v_2)$.
4. If $\psi(v_1) = \psi(v_2)$ for any $v_1, v_2 \in \mathbb{T}_1$, then either v_2 is a descendant of v_1, or vice versa. Furthermore, if u is a descendant of either v_1 or v_2, then $\psi(v_1) = \psi(u) = \psi(v_2)$.

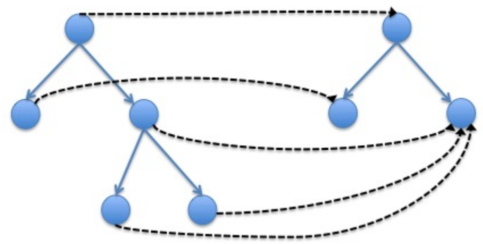


Figure 4: Production-Edit Tree Mapping: (left) $\mathbb{T}_{OPT} \to \mathbb{T}$ (right)

Note then that if $v_1 \neq v_2$ then $\psi(v_1) = \psi(v_2)$ can only occur if $\psi(v_1) = \psi(v_2)$ is a leaf node in $\mathbb{T}$. For any path $\mathcal{P}$ in $\mathbb{T}_1$, such an mapping is injective on the vertices of $\mathcal{P}$ up until a certain vertex v, possibly the last, whereafter all vertices in the subtree rooted at v are mapped to a single vertex of $\mathbb{T}_2$ which is a leaf node. The reason for this property is that there will be a case in our argument where it will be necessary to send the entire subtree rooted at node $v \in \mathbb{T}_{OPT}$ to a single leaf $u \in \mathbb{T}$, corresponding to nullifying a nonterminal. This can be visualized as a *trimming* of the production-edit tree $\mathbb{T}_1$.

We are now ready to explicitly construct a tree $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$, and the mapping which maps $\mathbb{T}_{OPT}$ into it. Explicitly, for a mapping ψ, we would like the starting and ending clouds of the path given by any node $v \in \mathbb{T}_{OPT}$ to be as close as possible to those of $\psi(v)$. If we can place bounds on this distance for all pairs $v, \psi(v)$, then we show how to place bounds on the total cost of all vertices in $\mathbb{T}$. The following theorem, Theorem 8, does exactly this. For the purpose of the theorem, we will need to introduce some important notation.

- *Notation for mapping & nodes:* Let ψ be a mapping from $\mathbb{T}_{OPT}$ to $\mathbb{T}$. Let $[X_{i-1}, Y_{i-1}, c_{i-1}] \to [X_i, Y_i, c_i]$ be any non-leaf node and its child in $\mathbb{T}_{OPT}$ that **are not** both mapped to the same node of $\mathbb{T}$ by ψ.
- *Notation for starting & ending clouds:* Let $(p_i, q_i), (r_i, s_i) \in \mathcal{T}$ be the starting and ending clouds of $[X_i, Y_i, c_i]$, and let $(p_{i-1}, q_{i-1}), (r_{i-1}, s_{i-1}) \in \mathcal{T}$ be the starting and ending clouds of $[X_{i-1}, Y_{i-1}, c_{i-1}]$. Similarly, let $(p'_i, q'_i), (r'_i, s'_i)$ and $(p'_{i-1}, q'_{i-1}), (r'_{i-1}, s'_{i-1})$ be the starting and ending clouds of $\psi([X_i, Y_i, c_i])$ and $\psi([X_{i-1}, Y_{i-1}, c_{i-1}])$ respectively. Define $(p_L, q_L), (p_R, q_R)$ to be the starting clouds of the left and right children of $[X_{i-1}, Y_{i-1}, c_{i-1}]$, respectively, and $(p'_L, q'_L), (p'_R, q'_R)$ to be the starting clouds of the left and right children of

$\psi([X_{i-1}, Y_{i-1}, c_{i-1}])$ respectively. Similarly, we denote their corresponding ending clouds by $(r_L, s_L), (r_R, s_R)$ and $(r'_L, s'_L), (r'_R, s'_R)$.

- *Notation for costs:* Let c'_i be the cost of $\psi([X_i, Y_i, c_i]) \in \mathbb{T}$. Write c_L, c_R for the costs of the left and right children of $[X_{i-1}, Y_{i-1}, c_{i-1}]$, and similarly write c'_L, c'_R for the costs of the children of $\psi([X_{i-1}, Y_{i-1}, c_{i-1}])$. For any cost c corresponding to a node v, let $\bar{c}$ denote the cost of all nodes in the subtree rooted at v if $v \in \mathbb{T}_{OPT}$, or the cost of all nodes mapped to v if $v \in \mathbb{T}$.

Given a node $[X_{i-1}, Y_{i-1}, c_{i-1}]$, the proof of the Theorem will have several cases when constructing the children of $\psi([X_{i-1}, Y_{i-1}, c_{i-1}])$. In each case, different bounds will hold, and these bounds will be vital to the proof of Theorem 9.

Theorem 8. *For any approximation algorithm $\mathcal{A} \in \mathcal{F}$ with precision functions α, β, there exists a tree $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$ and a Production-Edit Tree Mapping $\psi : V(\mathbb{T}_{OPT}) \to V(\mathbb{T})$ such that:*

Depending on whether $\psi([X_i, Y_i, c_i])$ is a leaf or Non-leaf, one of the following two holds:

$$c'_i \leq c_i + (|p'_i - p_i| + |q'_i - q_i|) - (|r'_i - r_i| + |s'_i - s_i|) + 2\alpha(r_i, s_i) \qquad \text{(Non-leaf)}$$

$$c'_i \leq \overline{c_i} + |p'_i - p_i| + |q'_i - q_i| + \beta(r_{i-1}, s_{i-1}) \qquad \text{(Leaf)}$$

Furthermore, depending on how the children of $\psi([X_{i-1}, Y_{i-1}, c_{i-1}])$ are constructed, either:

$$(|p'_L - p_L| + |q'_L - q_L|) + (|p'_R - p_R| + |q'_R - q_R|) \leq |r'_{i-1} - r_{i-1}| + |s'_{i-1} - s_{i-1}| + 2\beta(r_{i-1}, s_{i-1}) \quad (*)$$

Or one of the following inequalities holds as an upper bound for $c'_L + c'_R$

$$\leq \overline{c_L} + c_R + |r'_{i-1} - r_{i-1}| + |s'_{i-1} - s_{i-1}| + 2\beta(r_{i-1}, s_{i-1}) \qquad (**)$$

$$\leq c_L + \overline{c_R} + |r'_{i-1} - r_{i-1}| + |s'_{i-1} - s_{i-1}| + 2\beta(r_{i-1}, s_{i-1}) \qquad (**)$$

$$\leq \overline{c_L} + c_R + |r'_{i-1} - r_{i-1}| + |s'_{i-1} - s_{i-1}| + 2\beta(r_{i-1}, s_{i-1}) - (|r'_R - r_R| + |s'_R - s_R|) + 2\alpha(r_R, s_R) \quad (***)$$

$$\leq c_L + \overline{c_R} + |r'_{i-1} - r_{i-1}| + |s'_{i-1} - s_{i-1}| + 2\beta(r_{i-1}, s_{i-1}) - (|r'_L - r_L| + |s'_L - s_L|) + 2\alpha(r_L, s_L) \quad (***)$$

Before we prove Theorem 8, we show how our main theorem follows from it. 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 $\psi(v)$, or v is the node closest to the root that is mapped to $\psi(v)$. For the following proof of Theorem 9, it will be helpful to notice that the tree $\mathbb{T}'_{OPT}$ is the result of "trimming" off all the extra nodes in $\mathbb{T}_{OPT}$, and is in fact isomorphic to $\mathbb{T}$ as a graph.

Theorem 9. *For any $\mathcal{A} \in \mathcal{F}$ with precision functions α, β, let $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$ and ψ be as constructed in Theorem 8, and label the nodes of $\mathbb{T}'_{OPT}$ by $v_1 \dots v_K$. For $1 \leq i \leq K$, let $(p_i, q_i), (r_i, s_i)$ be the starting and ending clouds of the path v_i in $\mathcal{T}$, and let $(p'_i, q'_i), (r'_i, s'_i)$ be the starting and ending clouds of $\psi(v_i)$. Then*

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

Proof. The above bound will be the result of summing the various bounds (Non-leaf), (Leaf), $(**)$, and $(***)$ from Theorem 8 over all the v_j's (for each node, the bound we use depends on which of the cases the node falls into). For any node $u \in \mathbb{T}$, the cases are:

- If u is a non-leaf node that was constructed in Case 1 of Theorem 8, then there a unique node $v \in \mathbb{T}_{OPT}$ for which $\psi(v) = u$. In this case the bound (Non-leaf) from Theorem 8 is used.

- If u is a leaf node that was constructed in Case 1 of Theorem 8, then again there is a unique node $v \in \mathbb{T}_{OPT}$ for which $\psi(v) = u$, and we can apply the bound (Leaf).
- If u is a leaf node constructed in Case 2 of Theorem 8, let $w \in \mathbb{T}$ be the other child of the parent of u. Then if w is a leaf node, the bound (**) is applied to the costs of the nodes u and w. If w is not a leaf then the bound (***) is applied to the costs of u and w.

Note that this covers all cases, since every time a pair of children is constructed in Case 2 at least one of the children must be a leaf. Now if u is a leaf node constructed in Case 2, then there can be multiple nodes in $\mathbb{T}_{OPT}$ that are mapped to it. This can only occur when there is some unique vertex v which maps to u, such that all the descendants of v also map to u, and no other vertices in $\mathbb{T}_{OPT}$ can map to u. If this occurs then the corresponding bound, either $(**)$ or $(***)$, includes the sum of the costs of all nodes in the subtree rooted at $v \in \mathbb{T}_{OPT}$. Then such a v is in $\mathbb{T}'_{OPT}$, so the bounds given above considered for all nodes in $\mathbb{T}'_{OPT}$ includes the cost of all nodes in $\mathbb{T}_{OPT}$ which are not in $\mathbb{T}'_{OPT}$, thus we can consider only the bounds given by the nodes in $\mathbb{T}'_{OPT}$.

We now define subsets U^1_{NL}, U^2_{NL}, U_L of $\mathbb{T}'_{OPT}$.

- Let U^1_{NL} be the set of non-leaf nodes $v \in \mathbb{T}'_{OPT}$ where both children of $\psi(v) \in \mathbb{T}$ are constructed in Case 1 of Theorem 8. Note that the node in U^1_{NL} may be created either in Case 1 or Case 2.
- Let U^2_{NL} be the set of non-leaf nodes in $v \in \mathbb{T}'_{OPT}$ where the children of $\psi(v)$ are constructed in Case 2 of Theorem 8.
- Let U_L be the set of leaves $v \in \mathbb{T}'_{OPT}$ such that $\psi(v)$ was constructed in Case 1 of Theorem 8.
- The set of leaves v such that $\psi(v)$ was constructed in Case 2 will not be considered because the bounds for these nodes will be included in the bounds given by the parents in U^2_{NL}.

Note that for any $v_j \in \mathbb{T}'_{OPT}$, v_j is a leaf iff $\psi(v_j)$ is a leaf. Then for any $v_j \in U^1_{NL} \cup U^2_{NL}$, let R_j, L_j be the indices of its right and left children v_{R_j}, v_{L_j}. For any non-root node $v_j \in \mathbb{T}'_{OPT}$, let P_j be the index of its parent v_{P_j}. Since ψ is bijective when restricted to $\mathbb{T}'_{OPT}$, by summing the bounds (Non-leaf), (Leaf), $(**)$, and $(***)$ over all nodes in $\mathbb{T}'_{OPT}$, we are summing the bounds for all nodes in $\mathbb{T}$ as well. Summing over the costs given by these bounds in 8 for all nodes in $\mathbb{T}$, we have

$$\|\mathbb{T}\| \leq \|\mathbb{T}_{OPT}\| + \mathcal{W}^1_{NL} + \mathcal{W}^2_{NL} + \mathcal{W}_L$$

The rest of the proof will be spent analyzing these new terms. Firstly, $\mathcal{W}^1_{NL} =$

$$\sum_{v_j \in U^1_{NL}} \Big(|p'_{R_j} - p_{R_j}| + |q'_{R_j} - q_{R_j}| + |p'_{L_j} - p_{L_j}| + |q'_{L_j} - q_{L_j}| - (|r'_j - r_j| + |s'_j - s_j|) + 2\alpha(r_j, s_j)) \Big)$$

$$\mathcal{W}^2_{NL} = \sum_{v_j \in U^2_{NL}} \Big((|r'_j - r_j| + |s'_j - s_j|) + 2\beta(r_j, s_j) - (|r'_j - r_j| + |s'_j - s_j|) + 2\alpha(r_j, s_j)) \Big)$$

$$\mathcal{W}_L = \sum_{v_j \in U_L} \Big(\beta(r_{P_j}, s_{P_j}) \Big)$$

First sum: We first consider $\mathcal{W}^1_{NL}$. For each node $v_j \in U^1_{NL}$, if v_j *was not* created in Case 2 of Theorem 8 then the bound (Non-leaf) applies, and the bound $(*)$ applies to the children of v_j since they were also not created in Case 2. If v_j *was* created in Case 2, then one of the bounds $(***)$ will hold for the cost of v_j. Note in the second case that the sibling node of v_j must be a leaf. In either case we have put the portion $\big(-(|r'_j - r_j| + |s'_j - s_j|) + 2\alpha(r_j, s_j)$ of the bound for v_j next to the positive portions of the bounds which come from its children (either (Non-leaf) or (Leaf)) $|p'_{R_j} - p_{R_j}| + |q'_{R_j} - q_{R_j}| + |p'_{L_j} - p_{L_j}| + |q'_{L_j} - q_{L_j}|$ in the same summand. The goal of doing this is to easily apply Property $(*)$ of Theorem 8, which gives $|p'_{R_j} - p_{R_j}| + |q'_{R_j} - q_{R_j}| + |p'_{L_j} - p_{L_j}| + |q'_{L_j} - q_{L_j}| - (|r'_j - r_j| + |s'_j - s_j|) \leq 2\beta(r_j, s_j)$. Thus the bound on the cost of any node v_j is split between the summands corresponding to v_j and the summands corresponding to its parent in $\mathbb{T}_{OPT}$. If this bound was (Non-leaf), then the rest of the bound for v_j will be put in the sum $\mathcal{W}^1_{NL}$ for the parent of v_j. If the bound was one of the $(***)$ bounds, then the rest of the bound will be put in the sum $\mathcal{W}^2_{NL}$ for the parent of v_j. Note in the special case of the root we have $|p'_1 - p_1| + |q'_1 - q_1| = 0$, and so this does not appear in the sum $\mathcal{W}^1_{NL}$. Then using Property $(*)$, gives

$$\mathcal{W}^1_{NL} \leq \sum_{v_j \in U_{NL}} \Big(2\alpha(r_j, s_j) + 2\beta(r_j, s_j)\Big)$$

Second sum: We now consider $\mathcal{W}^2_{NL}$. For each node $v_j \in U^2_{NL}$, the bound (Non-leaf) applies, and the bound $(**)$ applies to the children of v_j since they *were* created in Case 2. Again, we have put the portion $\big(-(|r'_j - r_j| + |s'_j - s_j|) + 2\alpha(r_j, s_j)$ of the bound (Non-leaf) for v_j next to the portion $\big(|r'_j - r_j| + |s'_j - s_j|\big) + 2\beta(r_j, s_j)$ of the bounds either $(**)$ or $(***)$ for the costs of the children of v_j. Canceling terms in the definition of $\mathcal{W}^2_{NL}$ above gives:

$$\mathcal{W}^2_{NL} = \sum_{v_j \in U^2_{NL}} \Big(2\beta(r_j, s_j) + 2\alpha(r_j, s_j))\Big)$$

Third sum: $\mathcal{W}_L$ simply accounts for the fact that for every leaf in the tree $\mathbb{T}$ that was not created in Case 2 of Theorem 8, the portion of the sum $(Leaf)$ that is included in the sum for its parent, either in $\mathcal{W}^1_{NL}$ or $\mathcal{W}^2_{NL}$, does not include the $\beta(r_{P_j}, s_{P_j})$ term. If $v \in \mathbb{T}'_{OPT}$ maps to a node $\psi(v)$ that was created in Case 2 of Theorem 8, then the bound on the cost of $\psi(v)$ was already included in the sum $\mathcal{W}^2_{NL}$, which is why we have not included these nodes in U_L. Then

$$\mathcal{W}_L = \sum_{v_j \in U_L} \Big(\beta(r_{P_j}, s_{P_j})\Big) \leq \sum_{v_j \in \mathbb{T}'_{OPT}} \beta(r_j, s_j)$$

So all together $\mathcal{W}^1_{NL} + \mathcal{W}^2_{NL} + \mathcal{W}_L \leq$

$$\sum_{v_j \in U^1_{NL}} \Big(2\alpha(r_j, s_j) + 2\beta(r_j, s_j)\Big) + \sum_{v_j \in U^2_{NL}} \Big(2\alpha(r_j, s_j) + 2\beta(r_j, s_j))\Big) + \sum_{v_j \in \mathbb{T}'_{OPT}} \beta(r_j, s_j)$$

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

This completes the proof.

□

For any $\mathcal{A}$, we write $|\mathcal{A}|$ to denote the edit distance returned by $\mathcal{A}$, and $|OPT|$ for the edit distance returned by an exact algorithm.

Theorem 10. *For any $\mathcal{A} \in \mathcal{F}$ with precision functions α, β, let $\mathbb{T}_{OPT}$ be the production-edit tree of any optimal algorithm. Label the nodes of $\mathbb{T}'_{OPT} \subset \mathbb{T}_{OPT}$ by $v_1 \dots v_K$. For $1 \leq i \leq K$, let*

$(p_i, q_i), (r_i, s_i)$ *be the starting and ending clouds of the path* v_i *in* $\mathcal{T}$*, and let* $(p'_i, q'_i), (r'_i, s'_i)$ *be the starting and ending clouds of* $\psi(v_i)$*. Then*

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

Proof. The result follows immediately from Theorems 7 and 9.

□

Approximation of Ultralinear Language Edit Distance: Theorem 10 allows us to place bounds on the additive approximation in terms of the structure of the optimum derivation tree. For each non-linear production made in such a derivation, another term is accumulated in the bound given in 10, thus in general the approximation performs better for derivations which have a high ratio of linear to non-linear productions. In particular, when the grammar G is k-ultralinear, we are able to provide more explicit bounds for algorithms in $\mathcal{F}$. Note that, as a result of the structure of ultralinear grammars, a production-edit tree for any k-ultralinear grammar can have depth at most k, as each time a non-linear production is made the resulting non-terminals are in a lower partition of Q, and there are k total partitions in Q.

We now give two specific approximation algorithms for k-ultralinear languages.

Theorem (1)**.** *If* $\mathcal{A}$ *is a* γ*-uniform grid approximation, then* $\mathcal{A}$ *produces a value* $|\mathcal{A}| = |\mathbb{T}_{\mathcal{A}}|$ *such that*

$$|OPT| \leq |\mathcal{A}| \leq |OPT| + O(2^k\gamma)$$

in $O(|P|(n^2 + (\frac{n}{\gamma})^3))$ *time.*

Proof. In this case, we have the upper bound $\alpha(p, q) = \beta(p, q) = 2\gamma$ for all $1 \leq i \leq k$ and $1 \leq p \leq q \leq n$. Since there can be at most 2^k vertices in a production-edit tree for any k-ultralinear grammar, it follows from Theorem 10 that:

$$|OPT| \leq |\mathcal{A}| \leq |OPT| + 2^k(10\gamma)$$

Runtime. We only compute non-linear production edges for $(n/\gamma)^2$ clouds. For each non-linear production, we compute corresponding to a substring of size m, at most $m/\gamma \leq n/\gamma$ break-points. Thus the total runtime is $O(|P|(\frac{n}{\gamma})^3)$ to compute non-linear edges, and $O(|P|n^2)$ to run a single source shortest path algorithm from the sink to all vertices of $\mathcal{T}^{\mathcal{A}}$, for a total runtime of $O(|P|(n^2 + (\frac{n}{\gamma})^3))$

□

Theorem (2)**.** *Let* $\mathcal{A}$ *be any* γ*-non-uniform grid approximation, then* $\mathcal{A}$ *produces a value* $|\mathcal{A}| = |\mathbb{T}_{\mathcal{A}}|$ *such that*

$$|OPT| \leq |\mathcal{A}| \leq |OPT| + O(k\gamma)$$

in $O\big(|P|\big(n^2 + \frac{n^3}{\gamma^2}\big)\big)$ *time.*

Proof. Let $V_i = \{v_1, \ldots v_l\}$ be the set of all vertices at depth i in $\mathbb{T}_{OPT}$, and let w_j be the substring corresponding to the ending cloud (r_j, s_j) of v_j for $1 \leq j \leq l$. Then $w_1, w_2, \ldots w_l$ are the substrings which our algorithm derives from nonterminals at depth lower than i in $\mathbb{T}_{OPT}$. If $\frac{n}{2^{t+1}} \leq |w_j| \leq \frac{n}{2^t}$, we have $(r_j, s_j) \in N_t$ and can therefore set the upper bound on the precision functions $\alpha(r_j, s_j) = \beta(r_j, s_j) = \frac{\gamma}{2^t}$. Since the substrings must be disjoint, clearly $\sum_{j=1}^{l} |w_j| \leq n$. Set $|v_j| = 2\alpha(r_j, s_j) + 3\beta(r_j, s_j)$. Then if $(r_j, s_j) \in N_t$, we have $|v_j| \leq 5\frac{\gamma}{2^t}$. Then $|v_j| \leq (2|w_j|)5\frac{\gamma}{n}$. We have

$$\sum_{v_j \in V_i} |v_j| \leq \sum_{j=1}^{l} 10|w_j|\frac{\gamma}{n} \leq 10(n)\frac{\gamma}{n} = 10\gamma = O(\gamma)$$

Note that this bound is independent of the depth i. Since any production-edit tree for a k-ultralinear grammar can have depth at most k, Theorem 10 states that the additive error is at most $\sum_{i=1}^{k}\sum_{v\in V_i}|v|$. As just shown, each inner sum is at most 10γ, thus the total additive error is $10k\gamma = O(k\gamma)$ as desired.

Runtime. Let $N_i'' \subset N_i' \subset N_i$ be the set of clouds which we actually construct non-linear edges for. There are at most $\frac{n}{2^{i+1}}$ L_j's in N_i, and each has at at most n clouds in it. Then if $2^i \leq \gamma$ we have:

$$|N_i''| \leq n\frac{n}{2^{i+1}}(\frac{2^i}{\gamma})^2 = \frac{n^2 2^{i-1}}{\gamma^2}$$

Now if $2^i > \gamma$, then we consider all clouds in N_i. In other words, if $j \leq \frac{n}{\gamma}$, then we consider all clouds in L_j. Now $|L_j| \leq n$ for all j, so we consider at most $\sum_{j=1}^{n/\gamma}|L_j| \leq \frac{n^2}{\gamma}$ in this second section. Thus in $\mathcal{T}^{\mathcal{A}}$ we create nonlinear edges for a total of

$$\sum_{i=1}^{\log(\gamma)}|N_i''| + \frac{n^2}{\gamma} \leq \sum_{i=1}^{\log(\gamma)}\frac{n^2 2^{i-1}}{\gamma^2} + \frac{n^2}{\gamma} \leq 2\frac{n^2}{\gamma}$$

clouds. Now since for every substring of length $\ell \leq \frac{n}{2^i}$, we consider at most $\ell\frac{2^i}{\gamma} \leq \frac{n}{\gamma}$ breakpoints the total number of breakpoints considered over all nodes in the first section (where $2^i \leq \gamma$) is at most $\frac{n}{\gamma}\sum_{i=1}^{\log(\gamma)}|N_i''| = \frac{n^3}{\gamma^2}$. For the second section with strings of length $\leq \frac{n}{\gamma}$, the number of breakpoints is at most

$$|L_1|(1) + |L_2|(2) + \dots |L_{n/\gamma}|\frac{n}{\gamma} = n(1) + (n-1)2 + \dots + (n - \frac{n}{\gamma})\frac{n}{\gamma} \leq n\sum_{i=1}^{n/\gamma} i = O(\frac{n^3}{\gamma^2})$$

Thus a total of $O(\frac{n^3}{\gamma^2})$ breakpoints are considered for each production in $|P|$ while constructing $\mathcal{T}^{\mathcal{A}}$. There are $O(|P|n^2)$ edges in $\mathcal{T}^{\mathcal{A}}$, thus running single source shortest path takes a total of $|P|n^2$ time, so the total runtime is $O(|P|(n^2 + \frac{n^3}{\gamma^2}))$.

□

5 Hardness Results

Linear Grammar Edit Distance Hardness A recent result of Backurs and Indyk [7] has demonstrated that finding a truly subquadratic algorithm for computing the exact edit distance between two binary strings would imply the falsity of the Strong Exponential Time Hypothesis (SETH). This result has been shown to hold even in the case where the input strings are binary [13]. We extend this SETH-hardness result by demonstrating that a truly subquadratic algorithm for linear language edit distance to a grammar of constant size would imply a truly subquadratic algorithm for binary string edit distance.

We first define the linear grammar $G = (Q, \Sigma, P, S)$ with nonterminals $Q = \{S, S_z\}$, alphabet $\Sigma = \{0,1\} \cup \{z\}$, and productions:

$$S \to 0S0 \mid 1S1| \mid S_z$$

$$S_z \to zS_z z \mid z$$

Given two binary strings, $A \in \{0,1\}^n$ $B \in \{0,1\}^m$, let B^R be the reverse of B. Then define:

$$w_{A,B} = A\overline{z}_{n+m}B^R$$

Where $\overline{z}_{n+m} = \oplus_{i=1}^{m+n} z$, and $\oplus$ is concatenation. Note that our grammar has constant size. The language it recognizes is precisely $\mathcal{L}(G) = \{C\overline{z}_t C^R \mid C \in \{0,1\}^k, k, t \geq 0\}$, which is the set of all binary palindromes separated by any number of z's in between. The purpose of the dummy variable z will be to avoid *cheating* in the editing procedure by blurring the line between which string is A and which is B^R in a valid editing procedure.

Theorem 11. *The language edit distance between $w_{A,B}$ and G is equal to the string edit distance between A and B.*

Proof. Let $d(w_{A,B}, G)$ be the language edit distance, and let $d(A, B)$ be the string edit distance between A and B. We first show $d(w_{A,B}, G) \leq d(A, B)$. Let $\tilde{e}_A$ be an *ordered* sequence of insertions, deletions, and substitutions which edit A into B with minimum cost. Then applying the editing procedure to the substring $w_{A,B}(1 : n)$ transforms $w_{A,B}$ into the string $B\overline{z}_{n+m}B^R$ with equal cost. Since $B\overline{z}_{n+m}B^R \in \mathcal{L}(G)$, we have $d(w_{A,B}, G) \leq d(A, B)$.

Now let $\tilde{e}_G$ be an *ordered* sequence of edits of minimum cost which modify $w_{A,B}$ into a valid word of $\mathcal{L}(G)$. We first argue that the substring $\overline{z}_{n+m}$ is not modified. Since $\mathcal{L}(G)$ admits a string $C\overline{z}_tC^R$ for any t, deleting any of the z's in $w_{A,B}$ would not decrease the Levenshtein distance between the input string and the language, *unless* potentially in the case that all $n + m$ of the z's were deleted. But clearly $D_{A,B} \leq n+m$, thus we need not consider this case. Similarly, inserting z's can never be helpful. Replacing one of the z's with a 0 or 1, or inserting a 0/1 within the substring $\overline{z}_{n+m}$, say at position j, would then force that all z's in position either $< j$ or $> j$ be deleted or replaced. We have established that deleting z's is never helpful, and the effect of replacing a string of z's from j to the ending of A or the start of B^R could be equivalently accomplished by inserting the same terminals between $\overline{z}_{n+m}$ and A or B^R respectively. Thus we can assume that the substring $\overline{z}_{n+m}$ is never modified by $\tilde{e}_G$.

Then $\tilde{e}_G$ can be partitioned into the set of edits made to the substring A, and the edits made to B^R. This gives a valid procedure to edit A into C and B into C for some $C \in \{0, 1\}^k$ and k. Since edit distance is a metric on the space of strings, we have $d(A, B) = \min_{k,C\in\{0,1\}^k} d(A, C)+d(C, B)$. But we have just shown that the left hand side is at most $d(w_{A,B}, G)$, which completes the proof. □

Theorem 12 (Hardness of Linear Language Edit Distance)**.** *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.*

Proof. The theorem follows immediately from 11 and from the results of [13]. □

Ultralinear Language Parsing Hardness A recent result of Abboud, Backurs and Williams [4] has shown that any algorithm which can solve the recognition problem for an input string of length n to a context free grammar $\mathcal{G}$ in time $O(n^F)$ can be modified to an algorithm which can solve the $3k$-clique problem on a graph of order n in time $O(n^{Fk})$. A well known conjecture of graph algorithms states that the smallest such value of F for which $3k$-clique can be solved is 3 for combinatorial algorithms, and ω for any algorithm, where ω is the exponent of fast matrix multiplication. A refutation of this conjecture would additionally result in faster exact algorithms for Max-Cut [54, 52], among other consequences.

The proof of hardness in [4] proceeds by enumerating all k-cliques in an input graph, and then judiciously constructing an input string w over an alphabet Σ which encodes all of these k-cliques. A grammar $\mathcal{G}$ of constant size is then introduced such that $\mathcal{G}$ accepts w if and only if the input graph contains a $3k$-clique.

In this section we adapt this approach so that the grammar in question is ultralinear. We do this by constructing an ultralinear grammar $\mathcal{G}_U^\ell$, parameterized by a constant ℓ, such that $\mathcal{L}(\mathcal{G}_U^\ell) \subset \mathcal{L}(\mathcal{G})$ and such that if w is a string constructed from a graph G as specified by [4], then G has a $3k$-clique if and only if $w \in L(\mathcal{G}_U^\ell)$. Our grammar is essentially $\mathcal{G}$, but with modifications made in order to bound the total number of non-linear productions which can be made during any derivation. Our grammar will have size $O(\ell) = O(n^3)$, but since $|w| \in O(k^2n^{k+1})$ and the blowup in grammar size is independent of k, this is not problematic. It follows that if the currently known clique algorithms are optimal, the recognition problem for ultralinear grammars cannot be solved in $o(Poly(|\mathcal{G}|)n^F)$ time, where F is as in the conjecture above. We present our adaptation $\mathcal{G}_U^\ell$ below.

Theorem 13 (Hardness of Ultralinear Grammar Parsing)**.** *There is a ultralinear grammar $\mathcal{G}_U^\ell = \mathcal{G}_U$ such that if we can solve the membership problem for string of length n in time $O(|\mathcal{G}_U|^\alpha n^c)$, where $\alpha > 0$ is some fixed constant, then we can solve the k-clique problem on a graph with n nodes in time $O(n^{c(k+3)+3\alpha})$.*

Encoding of the Graph Let G be a graph on n vertices. For every vertex $v \in V(G)$, let $\overline{v}$ be a unqiue binary encoding of v of size exactly $2\log(n)$. Let $N(v)$ be the neighborhood of v. We define a set of gadgets, which are exactly those introduced in [4], over the same alphabet $\Sigma = \{0, 1, \$, \#, a_{start}, a_{mid}, a_{end}, b_{start}, b_{mid}, b_{end}, c_{start}, c_{mid}, c_{end}\}$. Firstly are the so-called *node* and *list* gadgets:

$$NG(v) = \#\overline{v}\# \qquad LG(v) = \# \bigoplus_{u \in N(v)} (\$\overline{u}^R\$)\#$$

where $\overline{u}^R$ is the reverse of $\overline{u}$. We then enumerate all k-cliques in G, and use $\mathcal{C}_k$ to denote the set of all k-cliques in G. Let $t = \{v_1, \dots, v_k\} \in \mathcal{C}_k$ be any k-clique. Then the so-called "clique-node" and "clique-list" gadgets are given by

$$CNG(t) = \bigoplus_{v \in t} (NG(v))^k$$

$$CLG(t) = \Big(\bigoplus_{v \in t} LG(v)\Big)^k$$

Along with the additional three gadgets

$$CG_\alpha(t) = a_{start} CNG(t) a_{mid} CNG(t) a_{end}$$

$$CG_\beta(t) = b_{start} CLG(t) b_{mid} CNG(t) b_{end}$$

$$CG_\gamma(t) = c_{start} CLG(t) c_{mid} CLG(t) c_{end}$$

Finally, the encoding of the G into a string w is given by

$$w = (\bigoplus_{t \in \mathcal{C}_k} CG_\alpha(t)))(\bigoplus_{t \in \mathcal{C}_k} CG_\beta(t))(\bigoplus_{t \in \mathcal{C}_k} CG_\gamma(t)))$$

Note that $|w| \in O(k^2 n^{k+1})$, and the cost of constructing the string w is linear in its length.

The Ultralinear Grammar Our grammar $\mathcal{G}_U^\ell = (Q, \Sigma, P, S)$ is given by

$$Q = \Big(\bigcup_{i=1}^{\ell} \{\mathbf{V}^i_{\alpha\gamma}, \mathbf{V}^i_{\alpha\beta}, \mathbf{V}^i_{\beta\gamma}, \mathbf{S}^i_{\alpha\gamma}, \mathbf{S}^i_{\alpha\beta},\ \mathbf{S}^i_{\beta\gamma}, \mathbf{N}^i_{\alpha\gamma}, \mathbf{N}^i_{\alpha\beta}, \mathbf{N}^i_{\beta\gamma}\}\Big) \bigcup \{\mathbf{S}, \mathbf{S}^*_{\alpha\gamma}, \mathbf{S}^*_{\alpha\beta}, \mathbf{S}^*_{\beta\gamma}\mathbf{W}, \mathbf{W}', \}$$

where i ranges from $i = 1, 2, \dots, \ell$, for some ℓ which we will later fix. The main productions are:

$$\mathbf{S} \to \mathbf{W} a_{start} \mathbf{S}^1_{\alpha\gamma} c_{end} \mathbf{W} \qquad \mathbf{S}^*_{\alpha\gamma} \to a_{mid} \mathbf{S}^1_{\alpha\beta} b_{mid} \mathbf{S}^1_{\beta\gamma} c_{mid}$$

$$\mathbf{S}^*_{\alpha\beta} \to a_{end} \mathbf{W} b_{start} \qquad \mathbf{S}^*_{\beta\gamma} \to b_{end} \mathbf{W} c_{start}$$

Then for $xy \in \{\alpha\beta, \alpha\gamma, \beta\gamma\}$, and for $i = 1, 2, \ldots, \ell - 1$, we have the "$xy$-listing rules":

$$\mathbf{S}^i_{xy} \to \mathbf{S}^*_{xy} \qquad \mathbf{S}^i_{xy} \to \#\mathbf{N}^{i+1}_{xy}\$\mathbf{V}^{i+1}_{xy}\#$$
$$\mathbf{N}^i_{xy} \to \#\mathbf{S}^{i+1}_{xy}\#\mathbf{V}^{i+1}_{xy}\$ \qquad \mathbf{N}^i_{xy} \to \sigma\mathbf{N}^i_{xy}\sigma$$

Where $\sigma \in \{0, 1\}$. Finally, again for $i = 1, 2, \ldots, \ell - 1$, we have the "assisting rules":

$$\mathbf{W} \to \epsilon \mid \lambda\mathbf{W} \qquad \mathbf{W}' \to \epsilon \mid \sigma\mathbf{W}' \qquad \mathbf{V}^i_{xy} \to \epsilon \mid \$\mathbf{W}'\$\mathbf{V}^{i+1}_{xy}$$

For all $\lambda \in \Sigma$ and $\sigma \in \{0, 1\}$. Then for $i = 1, \ldots, \ell - 1$, the partition

$$Q_{2\ell} = \{\mathbf{S}\},\ Q_{2\ell-i} = \{\mathbf{S}^i_{\alpha\gamma}, \mathbf{N}^i_{\alpha\gamma}, \mathbf{V}^i_{\alpha\gamma}\},\ Q_\ell = \{\mathbf{S}^*_{\alpha\gamma}\}$$

$$Q_{\ell-i} = \{\mathbf{S}^i_{\alpha\beta}, \mathbf{S}^i_{\beta\gamma}, \mathbf{N}^i_{\alpha\beta}, \mathbf{N}^i_{\beta\gamma}, \mathbf{V}^i_{\alpha\beta}, \mathbf{V}^i_{\beta\gamma}\},\ Q_1 = \{\mathbf{S}^*_{\alpha\beta}, \mathbf{S}^*_{\beta\gamma}\}, Q_0 = \{\mathbf{W}, \mathbf{W'}\}$$

satisfies the ultra-linear property. Our grammar is the same as that in [4], except we replace the set of nonterminals $\{\mathbf{V}, \mathbf{S}_{\alpha\gamma}, \mathbf{S}_{\alpha\beta},\ \mathbf{S}_{\beta\gamma}, \mathbf{N}_{\alpha\gamma}, \mathbf{N}_{\alpha\beta}, \mathbf{N}_{\beta\gamma}\}$ by ℓ *identical* copies, each with a index in $\{1, \ldots, \ell\}$, such that every time one copy of a nonterminal in this set is produced from another via a non-linear production, the resulting copy has a strictly greater index. Note that we replace the $\mathbf{V}$ of [4] with 3 further copies $\{\mathbf{V}^i_{\alpha\gamma}, \mathbf{V}^i_{\alpha\beta}, \mathbf{V}^i_{\beta\gamma}\}$ for each $i = 1, \ldots, \ell$, such that $\mathbf{V}^i_{xy}$ can only be produced by $\mathbf{N}^{i-1}_{xy}, \mathbf{S}^{i-1}_{xy}$, and $\mathbf{V}^{i-1}_{xy}$ (the nonterminals with the same subscript), as opposed the $\mathbf{V}$ in [4] which could be produced by $\{\mathbf{N}_{xy}, \mathbf{S}_{xy}, V\}$ for any $xy \in \{\alpha\beta, \alpha\gamma, \beta\gamma\}$.

Finally, for every copy of such a nonterminal with index ℓ, we prevent this nonterminal from making any further non-linear productions. Doing this places a strict limit on the maximum number of times a given non-linear productions may be used, in order to preserve the ultralinear property. Since we have not added any new productions, but instead modified each non-linear production of [4] such that it cannot be used more than ℓ times, we have that $\mathcal{L}(\mathcal{G}^\ell_U) \subset \mathcal{L}(\mathcal{G})$. Thus, the language recognized by our grammar is strictly a subset of the language recognized by the context free grammar $\mathcal{G}$, which consists of strings which can be produced with arbitrarily many non-linear productions. Specifically, as $\ell \to \infty$, the language $\mathcal{L}(\mathcal{G}^\ell_U)$ becomes precisely $\mathcal{L}(\mathcal{G})$. Note that the number of nonterminals and the number of productions in $\mathcal{G}^\ell_U$ is linear in the size of $\mathcal{G}^\ell_U$, thus we have $|\mathcal{G}^\ell_U| = O(\ell)$

We will show that taking $\ell = O(n^3)$ will be sufficient in order to recognize any encoding w of a graph G which contains a $3k$-clique, which will prove 13. Our proof is essentially the same as that of [4], except we count the number of times that non-linear productions must be used to derive a string w which encodes a $3k$-clique.

Theorem 14. *Let w be an encoding of a graph G as given above. Then $\mathcal{G}^\ell_U \to w$ if and only if G contains a $3k$-clique.*

Proof. We first recall the listing rules, for $xy \in \{\alpha\beta, \alpha\gamma, \beta\gamma\}$ and $i = 1, 2, \ldots, \ell - 1$, they are:

$$(1)\ \mathbf{S}^i_{xy} \to \mathbf{S}^*_{xy} \qquad (2_{xy})\ \mathbf{S}^i_{xy} \to \#\mathbf{N}^{i+1}_{xy}\$\mathbf{V}^{i+1}_{xy}\#$$
$$(3_{xy})\ \mathbf{N}^i_{xy} \to \#\mathbf{S}^{i+1}_{xy}\#\mathbf{V}^{i+1}_{xy}\$ \qquad (4)\quad \mathbf{N}^i_{xy} \to \sigma\mathbf{N}^i_{xy}\sigma$$

where $\sigma \in \{0, 1\}$, and the last assisting rule

$$(5)\mathbf{V}^i_{xy} \to \epsilon \mid \$\mathbf{W}'\$\mathbf{V}^{i+1}_{xy}$$

The proof in [4] proceeds by following the productions of the $\mathcal{G}$, and demonstrating that any resulting string must satisfy certain properties. Furthermore, if w is an encoding the a graph G as specified above, w will have these properties if and only if G has a $3k$-clique

We prove our extension of their theorem by showing that any string corresponding to the encoding of a graph that is accepted by the original grammar $\mathcal{G}$, can be produced using the listing productions (2_{xy}) and (3_{xy}), and the assisting production (5), at most ℓ times for some ℓ that we will later fix. Since these are the only non-linear productions which can be used more than once in any derivation, this will demonstrate for such an ℓ that the CFG $\mathcal{G}_U^\ell$ will accept an encoding w of a graph G if and only if G has a $3k$-clique.

Our proof follows that of [4], where we consider the sequence of productions which must be taken in order to derive w. We can only begin by the production $S \to w_1 a_{start} \mathbf{S}_{\alpha\gamma} c_{end} w_2$, where a_{start} appears in $CG(t_\alpha)$ for some $t_\alpha \in (C_k)$ and c_{start} appears in $CG(t_\gamma)$ for some $t_\gamma \in \mathcal{C}_k$. From here, we must derive $\mathbf{S}_{\alpha\gamma} \to CNG(t_\alpha)\mathbf{S}_{\alpha\gamma}CLG(t_\gamma)$ before exiting $\mathbf{S}_{\alpha\gamma}$ via the production $\mathbf{S}_{\alpha\gamma} \to \mathbf{S}^*_{\alpha\gamma}$, after which we can no longer return to $\mathbf{S}_{\alpha\gamma}$. The only way to produce the string $CNG(t_\alpha)\mathbf{S}_{\alpha\gamma}CLG(t_\gamma)$ is via the so-called listing productions $(2), (3), (4)$, thus we can confine our attention to them.

Note that $CNG(t_\alpha)$ consists of $k^2 \leq n^2$ binary encodings of vertices in G, whereas $CLG(t_\gamma)$ consists of $k^2 n \leq n^3$ such encodings. The only way to derive elements on the left of $\mathbf{S}^i_{\alpha\gamma}$ is by using the second listing production (2) and then deriving them via $\mathbf{N}^i_{\alpha\gamma}$ using (4). Repeated use of (4) allows for the derivation of exactly one of the binary encodings in $CNG(t_\alpha)$, and its corresponding reverse in $CLG(t_\gamma)$, say the sequence $\overline{v}$. Then each time we use the second production we are able to derive exactly one out of all k^2 sequences in CNG. By repeatedly applying (5), the nonterminal $\mathbf{V}^i_{\alpha\gamma}$ produced along with $\mathbf{N}^i_{\alpha\gamma}$ can derive all the binary sequences on the right side of $\overline{v}^R \in LG(u) \in CLG(t_\gamma)$, for some $u \in G$, and the $\mathbf{V}^{i+1}_{\alpha\gamma}$ derived from (3), after $\mathbf{N}^i_{\alpha\gamma}$ completes the derivation of $\overline{v}$, can construct all such binary sequences on the left side. There are at most n such sequences in $LG(u)$, thus we need use the production (5) at most n times to derive the rest of the terminals in $LG(u)$.

Thus each time we derive one of the $LG(u)$'s in $CLG(t_\gamma)$, during which we simultaneously derive one of the $NG(v)$'s in $CNG(t_\alpha)$, we use at most n non-linear productions, thus increasing the index of any nonterminal by at most n. Note in actuality, since (5) only involves $\mathbf{V}^i_{\alpha\gamma}$, we only increase the index of $\mathbf{V}^i_{\alpha\gamma}$ by this much; the index of $\mathbf{N}^i_{\alpha\gamma}$ and $\mathbf{S}_{\alpha\gamma}$ increase by at most two, since both of (2) and (3) are used at most once in this process, but for simplicity we will use n as the upper bound. Since this process must be repeated at most k^2 times, the total increase in the indices of the nonterminals is at most $nk^2 \leq n^3$ in the derivation of $CNG(t_\alpha)\mathbf{S}_{\alpha\gamma}CLG(t_\gamma)$

Now once we have produced the sentential form $CNG(t_\alpha)\mathbf{S}^i_{\alpha\gamma}CLG(t_\gamma)$, the only possibility is to "exit" via the production $\mathbf{S}^i_{\alpha\gamma} \to \mathbf{S}^*_{\alpha\gamma}$ for some i. From here, we must apply the production $\mathbf{S}^*_{\alpha\gamma} \to a_{mid}\mathbf{S}^1_{\alpha\beta} b_{mid} \mathbf{S}^1_{\beta\gamma}$, and then seek to derive

$$\mathbf{S}^1_{\alpha\beta} \to CNG(t_\alpha)\mathbf{S}^*_{\alpha\beta}CLG(t_\beta) \qquad \mathbf{S}^1_{\beta\gamma} \to CNG(t_\beta)\mathbf{S}^*_{\alpha\beta}CLG(t_\gamma)$$

Again, as just argued, both of these derivations can be completed using at most n^3 non-linear productions, and thus never producing a nonterminal of index greater than n^3. Once this has occurred, the rest of the string w can be derived via the exiting productions $\mathbf{S}^*_{\alpha\beta} \to a_{end}\mathbf{W} b_{start}$ and $\mathbf{S}^*_{\beta\gamma} \to b_{end}\mathbf{W} c_{start}$, as W can produce any string in Σ^*. Since no nonterminal with index greater than n^3 is ever produced, by setting $\ell = 2n^3$ it follows that our grammar $\mathcal{G}_U$ will accept the string w via the previous productions.

Now it is proven explicitly in [4], that if the derivation $\mathbf{S}_{xy} \to CNG(t)\mathbf{S}^*_{xy}CLG(t')$ can occur using the only the xy-listing rules, for any $t, t' \in \mathcal{C}_k$ and $xy \in \{\alpha\beta, \alpha\gamma, \beta\gamma\}$, then the k-cliques t, t' must form a $2k$-clique $t \cup t'$. Since the set of all sentential forms, disregarding index, derivable from our grammar $\mathcal{G}_U^\ell$ is strictly a subset of its context free counterpart $\mathcal{G}$, this result immediately holds for $\mathcal{G}_U^\ell$ as well. Finally, since our derivation involved occurrences of all three of $\mathbf{S}_{\alpha\gamma} \to CNG(t_\alpha)\mathbf{S}_{\alpha\gamma}CLG(t_\gamma)$, $\mathbf{S}^1_{\alpha\beta} \to CNG(t_\alpha)\mathbf{S}^*_{\alpha\beta}CLG(t_\beta)$ and $\mathbf{S}^1_{\beta\gamma} \to CNG(t_\beta)\mathbf{S}^*_{\alpha\beta}CLG(t_\gamma)$, it follows that $t_\alpha \cup t_\beta \cup t_\gamma$ is a $3k$-clique.

The validity of the other direction can be demonstrated by following the derivations described above for any particular triple $t_\alpha, t_\beta, t_\gamma \in \mathcal{C}_k$ which together form a $3k$-clique, which completes the proof.

□

Theorem 13. We have shown that for $\ell = 2n^3$, our grammar $\mathcal{G}_U^\ell$ accepts the string w iff $\mathcal{G}$ contains a $3k$-clique. The size of our grammar is then $|\mathcal{G}_U^\ell| = O(\ell) = O(n^3)$. Since the size of the string w encoding the graph G was $O(k^2n^{k+1})$, which can be constructed in $O(k^2n^{k+1}) < O(n^{k+3})$ time, it follows that if the membership of $w \in \mathcal{L}(\mathcal{G}_U^\ell)$ can be determined in time $O(|\mathcal{G}_U^\ell|^\alpha n^c)$, then the $3k$-clique problem can be solved in time $O(n^{c(k+3)+3\alpha})$, which proves the theorem. $\square$

6 Metalinear and Superlinear Grammar Edit Distance

In this section we demonstrate a quadratic time algorithm for *metalinear* and *superlinear* grammars.

Definition (k-metalinear)**.** G is said to be k**-metalinear** if every production is of the form:

$$S \to A_1 \dots A_t$$

$$A_i \to \alpha A_j \beta$$

Where $A_i \in Q \setminus \{S\}$, $\alpha, \beta \in \Sigma^*$, and $t \leq k$.

Thus, a k-metalinear language can have at most k linear nonterminals on the right hand side of a production. The metalinear languages (also referred as $LIN(k)$) strictly contain the linear languages. Furthermore, it has been shown that $Lin(k)$ is a strict subset if $Lin(k+1)$ for every $k \geq 1$, giving rise to a infinite hierarchy within the metalinear languages [32].

Definition (superlinear)**.** G is said to be **superlinear** if there is a subset $Q_L \subset Q$ such that every nonterminal $A \in Q_L$ has only linear productions $A \to \alpha B$ or $A \to B\alpha$ where $B \in Q_L$ and $\alpha \in \Sigma$. If $X \in Q \setminus Q_L$, then X can have *non-linear productions* of the form $X \to AB$ where $A \in Q_L$ and $B \in Q$, or linear productions of the form $X \to \alpha A \mid A\alpha \mid \alpha$ for $A \in Q_L$, $\alpha \in \Sigma^*$. Superlinear grammars strictly contain the metalinear grammars.

Note that if we also allow both the nonterminals of the RHS to come from Q, then we get the entire class of context free grammars. A grammar G is superlinear iff every word $w \in L(G)$ can be expressed as the concatenation of words generated by linear grammars. This is a generalization of the metalinear languages, and can be thought of as the family $Lin(\infty)$. Superlinear grammars strictly contain the metalinear grammars, and are the regular closure of the linear languages. Several other nice properties of them have been well studied [32].

We now show how any metalinear grammar can be explicitly transformed into an equivalant superlinear grammar.

Conversion of Metalinear to Superlinear grammar. Let G_M be any k-linear grammar, we construct an equivalant superlinear grammar G_S. For every production of the form

$$S \to A_1 \dots A_t, \quad t \leq k$$

Add $t-1$ new nonterminals $A'_1, \dots A'_{t-1}$, and the following productions

$$S \to A'_{t-1} A_t$$

$$A'_i \to A'_{i-1} A_i \ \text{ for } i = 2, 3, \dots, t-1$$

$$A'_1 \to A_1$$

The result is a superlinear grammar G_S, with at most $p|P|$ new nonterminals, where p is the maximum number of nonterminals on the left hand side of any production. Under the assumption that $p = O(|P|)$, the $O(|P|n^2)$ time algorithm we present in this section for superlinear grammars gives an $O(|P|^2n^2)$ algorithm for metalinear language edit distance.

Algorithm. We now present a quadratic time-complexity algorithm for computing the minimum edit distance to a superlinear grammar G. Let $\overline{x} = x_1 \dots x_n$ be our input string. The algorithm has two phases.

First Phase. In the first phase, we construct a graph $\mathcal{T}^R$, which is precisely the linear grammar edit distance graph $\mathcal{T}(Q_L, \overline{x})$ for the nonterminals in Q_L, but with the direction of every edge reversed and the weights kept the same. This, in effect, switches the roles of the source and sink vertices of $\mathcal{T}$. Computing the single source shortest path starting from t, by the symmetry of $\mathcal{T}$ and $\mathcal{T}^R$ we obtain the weight of the shortest path from $A^{i,j}$ to t in $\mathcal{T}$ for every nonterminal $A \in Q_L$ and $1 \leq i \leq j \leq n$. By the proof of the correctness of the linear language edit distance algorithm (Theorem 5), the weight of such a path is equal to the minimum edit distance of $x_i \dots x_j$ to a string s which can be legally produced starting from the state A. Thus computing single source shortest path from t in $\mathcal{T}^R$ allows us to construct a matrix $T_{i,j}(A) = c$ such that c is the minimum cost of deriving $x_i \dots x_j$ from A. This, as before, can be done in $O(n^2|P|)$ time.

Second Phase. Once we have $T_{i,j}(A)$ computed for all i, j and $A \in Q_L$, we begin the second phase where we construct a new graph $\mathcal{T}_{NL}$ with a new sink vertex t_{NL}, NL for non-linear, consisting of n clouds, each of which has a vertex for each of the non-linear nonterminals $Q \setminus Q_L$. We will denote the *ith* cloud by (i), and for any non-linear nonterminal $A_k \in Q \setminus Q_L$, we denote the vertex corresponding to A_k in (i) by A_k^i. Cloud (i) will then correspond to the substring $x_i x_{i+1} \dots x_n$, and for any nonterminal $A_k \in Q \setminus Q_L$, the weight of the shortest path from A_k^i to t_{NL} will be equal to the minimum edit distance between $x_i x_{i+1} \dots x_n$ to the set of strings legally derivable from A_k. Thus the vertex set of the graph is given by: $V(T_{NL}) = \{A_k^i \mid A_k \in Q \setminus Q_L,\ 1 \leq i \leq n\} \cup \{t_{NL}\}$. Let $null(A)$ denote the length of the shortest string legally derivable from A. We show how this can be computed for any CFG in $O(|Q||P|\log(|Q|))$ time in Theorem 3. We now describe the construction of the edges of $\mathcal{T}_{NL}$.

Construction of the edges.

1. For every non-linear production $A_k \to BC$, and each $1 \leq i \leq j < n$, create the edge $A_k^i \xrightarrow[T_{i,j}(B)]{BC} C^{j+1}$. *$B$ derives the substring $x_i x_{i+1} ... x_j$ with a cost of $T_{i,j}(B)$*
2. For every non-linear production $A_k \to BC$ and each $1 \leq i \leq n$, create the edge $A_k^i \xrightarrow[null(B)]{BC} C^i$. *$B$ derives ϵ with a cost of $null(B)$*. Also create the edge $A_k^i \xrightarrow[T_{i,n}(B)+null(C)]{BC} t_{NL}$. *$B$ derives the substring $x_i x_{i+1} ... x_n$ with a cost of $T_{i,n}(B)$, and C derives ε with a cost of $null(C)$*
3. For each production $A_k \to B$, and each $1 \leq i \leq n$, create the edge $A_k^i \xrightarrow[T_{i,n}(B)]{B} t_{NL}$. *$B$ derives the substring $x_i x_{i+1} ... x_n$ with a cost of $T_{i,n}(B)$*

Theorem 15. *The weight W of the shortest path from S^1 to t_{NL} in $\mathcal{T}_{NL}$ is equal to the minimum language edit distance between $\overline{x}$ and G, and can be computed in $O(|P|n^2)$ time.*

Proof. The idea behind the proof is similar to that of the linear language edit distance algorithm. Every legal word $w \in L(G)$ can be derived starting from a string of nonterminals $A_1 A_2 \dots A_k$ (by suitable relabeling of the nonterminals) where $A_i \in Q_L$ for $1 \leq i \leq k$ and $S \to A_1 B_1 \to A_1 A_2 B_2 \to \dots \to A_1 A_2 \dots A_k$ with $B_i \in Q \setminus Q_L$.

Let $w = \bigoplus_{i=1}^k w_i$ be a partition of the word such that w_i is the substring derived by A_i. Then if e_w is any sequence of editing procedures (deletions of a terminal in w, insertions of a terminal into w, or replacement of a terminal in w) which edits w into $\overline{x}$ given a specified set of legal production p_w which produce w, then we show how e_w can be partitioned into $e_{w_1}, e_{w_2}, \dots, e_{w_k}$, where e_{w_i} are the edits of e_w restricted to the substring w_i. This partition works as follows. Let ϵ be any single edit. If it is a deletion of a terminal in w_i, or the replacement of a terminal in w_i with another terminal, then we put ϵ in e_{w_i}. If ϵ is an insertion of a terminal between two terminals a and b which are both either in w_i, the result of replacement of a terminal in w_i, or the result of an insertion edit ϵ' in e_{w_i}, then we put ϵ in e_{w_i}. If the insertion is made on the boundary, say where either $a \in w_i$, a was the result of a replacement of a terminal and w_i, or a was an insertion made in e_{w_i}, and either $b \in w_{i+1}$, b was the result of a replacement of a terminal and w_{i+1}, or b was an insertion made in $e_{w_{i+1}}$, then we assign ϵ to e_{w_i} (we could just as easily assign to $e_{w_{i+1}}$, as long as the rule is consistent). In other words, we assign ϵ to the substring w_i of the lowest index of the two substrings corresponding to the terminals on either side of the insertion ϵ.

Now fix any such string w and set of edits e_w with corresponding partition $e_{w_1}, e_{w_2}, \ldots, e_{w_k}$ such that e_w edits w into $\overline{x}$. Let $|e_{w_i}|$ be the total cost of the edits e_{w_i}, and let $\overline{x_i}$ be the result of applying e_{w_i} to w_i, then $\overline{x} = \bigoplus_{i=1}^{k} \overline{x_i}$. Now since the process of editing w_i into $\overline{x_i}$ is independent from the process of editing w_j into $\overline{x_j}$ for all $j \neq i$, the minimum edit distance from $\overline{x_i}$ to the set of strings that are legally derivable from A_i is less than or equal to the cost of e_{w_i} for all editing procedures e_{w_i}. By the proof of the linear grammar edit distance algorithm, for any $1 \leq a \leq b \leq n$ and nonterminal A_i, the value $T_{a,b}(A_i)$ is equal to the minimum edit distance between $x_a \ldots x_b$ and the set of strings legally derivable from A_i. Setting $l_i = \sum_{k=1}^{i-1} |\overline{x_k}|$, then we have in particular $T_{l_i+1, l_{i+1}}(A_i) \leq |e_{w_i}|$. Furthermore, for any sequence $1 = \ell_1 \leq \ell_2 \leq \cdots \leq \ell_{k+1} = n$, the path:

$$S^1 \xrightarrow[T_{\ell_1,\ell_2}(A_1)]{A_1 B_1} B_1 \xrightarrow[T_{\ell_2+1,\ell_3}(A_2)]{A_2 B_2} B_2 \longrightarrow \ldots \longrightarrow t_{NL}$$

exists in $\mathcal{T}_{NL}$ with cost $\sum_{i=1}^{k} T_{\ell_i,\ell_{i+1}}(A_i)$. Setting $\ell_i = l_i$, it follows that $\sum_{i=1}^{k} T_{\ell_i,\ell_{i+1}}(A_i) \leq |e_w|$ for any editing procedure which transforms a string w derivable from $A_1 \ldots A_k$ into $\overline{x}$. In particular, this holds for the editing procedure with minimum cost, from which we conclude that W is at most the minimum language edit distance from $\overline{x}$ to G.

The fact that W can be no less than the minimum edit distance is easily seen, as every path corresponds to a derivation $S \to A_1 B_1 \to A_1 A_2 B_2 \to \cdots \to A_1 A_2 \ldots A_k$, and a partition $\overline{x_1}, \overline{x_2}, \ldots, \overline{x_k}$ such that $\overline{x} = \bigoplus_{i=1}^{k} \overline{x_i}$ and the cost of the path is the sum of the minimum costs of editing $\overline{x_i}$ into a string legally derivable from A_i over all $1 \leq i \leq k$. If W were less then the optimal, then the shortest path on $\mathcal{T}_{NL}$ would give a string of nonterminals $A_1 \ldots A_k$ derivable from S such that $\overline{x}$ can be edited into a string legally derivable from $A_1 \ldots A_k$ with cost less than the language edit distance, a contradiction, which completes the proof.

Running Time. The first phase of the algorithm takes time $O(|P|n^2)$, as it entails running single source shortest path on the linear grammar edit distance graph $\mathcal{T}$. The graph $\mathcal{T}_{NL}$ constructed in the second phase has $O(|Q|n)$ vertices, and $O(|P_A|n)$ edges connecting to any vertex $A^i \in \mathcal{T}_{NL}$, where $P_A \subset P$ is the subset of productions with A on the left hand side. Thus the total number of edges is $O(|P|n^2)$, so running single source shortest path on a graph takes $O(|P|n^2)$ time, therefore the entire algorithm runs in $O(|P|n^2)$ time. ◻

Theorem 16. *The language edit distance to any metalinear grammar can be computed in $O(|P|^2 n^2)$ time.*

Proof. Follows directly from the above theorem and the conversion of any metalinear grammar to superlinear grammar. ◻

7 Appendix

7.1 Proof of Theorem 4

Proof. Let ρ be a sequence of legal productions which derives a string s from A_k, interspersed by d edits that edit s into $x_i \ldots x_j$, where d is the optimal edit distance over the set of all such strings s derivable from A_k. We first show that the shortest path has weight d.

Base Case $d = 0$. If $d = 0$, then starting from the vertex $A_k^{i,j}$, we can follow edges for *legal productions* and *completing legal productions* according to ρ of weight 0 to reach t. (For example, if the first production in ρ is $A_k \to x_i A_{k_1}$, then the first edge taken will be $A_k^{i,j} \xrightarrow[0]{x_i} A_{k_1}^{i+1,j}$.)

Base Case $d = 1$. Let us now consider $d = 1$. The single error has caused by either substitution, or insertion, or deletion.

single substitution error. First, consider when a single substitution error has happened at position l. That is, if we had replaced x_l by some $a \in \Sigma$, the substring $s = x_i x_{i+1} ... x_{l-1} a x_{l+1} ... x_j$ can be derived from A_k with cost $d = 0$. Consider the series of legal productions made from A_k, up until the point where a is produced. At this point, there is some string $x_\ell \dots a$, or $a \dots x_{\ell'}$ which is left to be derived by some non-terminal $A_t \in Q$. WLOG, the string $x_\ell \dots a$ remains to be derived. For all the productions so far, we follow the edges created for *legal productions*, giving us a path from $A_k^{i,j}$ to $A_t^{\ell,l}$ of cost 0. At this point, we can take the replacement edge $A_t^{\ell,l} \xrightarrow[1]{x_l} A_q^{\ell,l-1}$, where $A_t \to A_q a$ is a production, or if $\ell = l$ then $A \to a$ is a production, so we take $A_t^{\ell,\ell} \xrightarrow[1]{x_l} t$. In the second case we are done. In the first we have the string $x_\ell \dots x_{l-1}$ left to derive from A_q, which can be done with cost 0 by again following the legal production edges, corresponding to the productions of ρ, to the sink. The concatenation of the paths from $A_k^{i,j}$ to $A_t^{\ell,l}$, of cost 0, then the edge of cost 1 to $A_q^{\ell,l-1}$, and then the path of cost 0 from $A_q^{\ell,l-1}$ to the sink, gives a full path from $A_k^{i,j}$ to t with a total cost of 1.

single deletion error. Now consider a single deletion error at position l. Hence, $s = x_i x_{i+1} ... x_{l-1} a x_l x_{l+1} ... x_j$ can be derived from A_k with $d = 0$. Then follow the series of legal productions of ρ until a is produced. At this point, we must either derive $x_\ell \dots x_{l-1} a$ or $a x_l \dots x_{\ell'}$ from a non-terminal A_t. WLOG $x_\ell \dots x_{l-1} a$ remains. Again, follow the legal edges from $A_k^{i,j}$ to $A_t^{\ell,x_{l-1}}$ with cost 0. Then take the edge $A_t^{\ell,l-1} \xrightarrow[1]{\epsilon(a)} A_q^{\ell,l-1}$, where $A_t \to A_q a$ is a production. Starting from $A_q^{\ell,l-1}$, we again follow the remaining legal edges, corresponding to the remaining productions in ρ which produce the rest of the string, to the sink. The whole path all together, then, takes us from $A_k^{i,j}$ to the sink with cost 1 as desired. One final case occurs if a is the last terminal derived in the sequence. Then either x_{l-1} or x_l was the last terminal derived when we are stopped, WLOG it is x_l with the production $A_r \to A_s x_l$. Then the last production of ρ must be a production $A_s \to a$. Then we have $null(A_s) = 1$. Then we can take legal edges from $A_k^{i,j}$ to $A_r^{l,l}$ with cost 0. We then take the edge $A_r^{l,l} \xrightarrow[null(A_s)]{x_l} t$, giving a full path to the sink with cost 1 as desired.

single insertion error. Now consider a single insertion error at position l. Hence, $s = x_i x_{i+1} ... x_{l-1} x_{l+1} ... x_j$ can be derived from A_k with $d = 0$. Again, consider the sequence of legal productions made until either x_{l-1} or x_{l+1} is derived, whichever happens first. WLOG x_{l-1} is derived first by a non-terminal $A_t \to x_{l-1} A_q$, with $x_{l+1} \dots x_\ell$ left to be derived from A_q. Then follow the corresponding legal production edges from $A_k^{i,j}$ to $A_q^{l,\ell}$, and then take the insertion edge $A_q^{l,\ell} \xrightarrow[1]{x_l} A_q^{l+1,\ell}$. From here, follow the edges given by the remaining legal productions of ρ, which takes us from $A_q^{l+1,\ell}$ to the sink with cost 0. Then the whole path has cost 1, as desired.

Induction. Assuming the result is true for errors up to $d - 1$, the induction step for d edits is easy. Let s be the legally derivable string. Consider the sequence of legal productions in the production sequence of s up until either the production of a terminal which will be deleted, or a terminal which will be substituted, or to the point where a terminal will need to be inserted. let $x_i x_{i+1} \dots x_{r-1}$ and $x_{s+1} \dots x_j$ be the substrings that were derived by these legal productions so far at the point when we are stopped. Taking the corresponding legal production edges gives a path of cost 0 from $A_k^{p,q}$ to $A_t^{r,s}$ for some $A_t \in Q$. Now, in if the case is substitution, we apply the argument from the base case and arrive at a vertex $A_q^{r+1,s}$ or $A_q^{r,s-1}$ with cost 1, WLOG we are at $A_q^{r+1,s}$. Now there are $d - 1$ remaining edits in between the string left to be legally derived by A_q and the string $x_{r+1} \dots x_s$. Thus the induction hypothesis applies, and we obtain a path of weight $d-1$ from $A_q^{r+1,s}$ to the sink. Concatenating the paths gives the desired path of length d.

In the case of deletion or substitution, we similarly follow the argument in the base case for $d = 1$, and then apply the induction hypothesis on the remaining substring left to be derived. The only distinct case to note is in the deletion case, if we are stopped at a point where we have derived all terminals in x except x_l, and there remains to be derived the substring $x_l a_1 a_2 \dots a_m$ of s – meaning that all of $a_1 \dots a_m$ must be deleted. In this case, we make the same argument at the beginning of the deletion base case, taking a path of cost 0 from $A_k^{i,j}$ to $A_r^{l,l}$. Suppose the next step in the derivation is $A_r \to x_l A_s$. Then we take the null edge $A_r^{l,l} \xrightarrow[null(A_s)]{x_l} t$ with cost at most m since $a_1 \dots a_m$ can be derived from A_s. Note that the cost must be exactly m, since the edit distance d is assumed to be optimal.

This completes all cases, thus the shortest path has weight at most d. Let d^* be the weight of the shortest path. Then reversing the process of reasoning taken above, any such path from $A_k^{i,j}$ to t of weight d^* gives rise to a production of s from A_k which can then be edited into $x_i \dots x_j$ using exactly d^* edits. Since d is the optimal edit distance, we have that $d^* = d$ is the weight of the shortest path as desired. Furthermore, considering the other direction, this means that if d^* is the weight of the shortest path, then the minimum edit distance must also be d^*, which completes the proof.

□

7.2 Proof of Theorem 3

Proof. The problem of finding $null(A)$ is solved by the algorithm given in [30]. Specifically, the problem of finding the shortest string derivable from A is application **(B)** of [30]. The algorithm takes as input a context free gramamr G and nonterminal $A \in Q$, and returns the value $null(A)$ in time $|P|\log(|Q|) + |P| = O(|P|\log(|Q|))$. Repeating the process for all nonterminals in $|Q|$ yields the desired runtime. □

7.3 Proof of Theorem 6

Proof. Note in this proof we work with the graphs $\overline{L_i}$ instead of $\overline{L_i^R}$ for simplicity. Since each path in one is just a reversal of the other, this is a trivial modification.

First, for any string $x \in \Sigma^*$, we write $x(p:q)$ to denote the substring $x_p x_{p+1} \dots x_q$. The proof is by induction on i. For $A^{\ell,\ell \in} \overline{L}_1$, consider the optimal editing procedure O from x_ℓ to the set of strings derivable from A. If this optimal edititing procedure does not involve a non-linear production, then the result follows from Theorem 4. If O does use a non-linear production, then it must be a production which nullifies a resulting non-terminal (Step. 1). This then corresponds to using a *null* edge created by our algorithm. Then necessarily O makes a series of non-linear productions, each time nullifying exactly one of the two resulting nonterminals, until we reach a nonterminal A_* from which we take a linear edge (possibly an error edge). The minimum cost of doing this is given by the *null* function, and thus following the corresponding null edges created in step 1 gives a path from $A^{\ell,\ell}$ to $A_*^{\ell,\ell}$. This cost of this path is precisely the optimal cost of nullifying all specified nonterminals. From A_* only linear productions are made, thus the cost of the shortest path from $A_*^{\ell,\ell}$ to t is precisely the cost of the remaining productions in O by Theorem 4.

Now assume the result for $1, \dots, i-1$, and fix any $A^{p,q} \in \overline{L}_i$, noting that necessarily $q - p + 1 = i$, and consider an optimal series of legal and illegal (error) productions which produce $x(p:q)$ from A (note that every error production is a linear production). There are three cases, and consider the first production in this series. There are three cases:

If the first production is to derive x_p either via an insertion, replacement, or valid production, then this corresponds to a unique edge $A^{p,q} \to B^{p+1,q}$ with cost $\gamma \in \{0,1\}$, where γ depends on whether or not this production was an error. Suppose this edge takes us to $B^{p+1,q} \in \overline{L_{i-1}}$. In step 2, we create an edge from t to $A^{p,q}$ of cost $\gamma + T_{p+1,q}(B)$. By induction, $T_{p+1,q}(B)$ is the minimum edit distance between $x(p+1:q)$ and the set of strings which can be legally produced from B. Thus the cost of this edge is indeed the minimum edit distance between $x(p:q)$ and the set of strings which can be legally produced from A. The same argument holds when the terminal in question is x_q.

If the first production is a non-linear production $A \to BC$, and B and C each produce at least one terminal of $\overline{x}$, then it must be the case that there is some optimal splitting point $p \le \ell < q$ such that B derives $x(p:\ell)$ with cost c_1 and C derives $x(\ell+1,q)$ with cost c_2. Since each of these substrings is strictly smaller than i, they each correspond to a cloud in $\{\overline{L_1}, \dots, \overline{L_{i-1}}\}$, and since step 3 of the algorithm creates an edge with cost c which is at most $T_{p,\ell}(B) + T_{\ell+1,q}$ (since c is computed as minimum over all splitting points), by induction we know $c \le T_{p,\ell}(B) + T_{\ell+1,q} \le c_1 + c_2$. Since both c_1 and c_2 must necessarily be optimal costs of deriving $x(p,\ell)$ from B and $x(\ell+1,q)$ from C respectively, the cost of the edge created in step 3 is precisely $c_1 + c_2$. Thus taking this edge gives a shortest path which is indeed equal to the minimum edit distance between the substring $x_p \dots x_q$ and the set of strings which can be legally produced from A.

Finally, we consider the case that first production is a non-linear production $A \to BC$, and one of B or C creates no terminals in $\overline{x}$ (is nullified). WLOG, B is the nullified nonterminal. The corresponding edges are constructed in step 4, and takes us to $C^{p,q}$ with cost $null(A)$. By theorem 3, we can correctly compute $null(B)$ prior to commencement of the algorithm.

Now any edge taken from $A^{p,q}$ correspond to a derivation of $x(p:q)$ from A using both legal and error productions. Since the cost of these edges corresponds to the cost of the derivation, it must be the case that $T_{i,j}(A)$ is no less than the minimum edit distance between $x(p:q)$ and the set of strings which can be legally produced from A, which completes the proof.

□

7.4 Proof of Theorem 7

Proof. If $\mathcal{A}$ returns c, then c is the length of the shortest path from $S^{1,n} \in \overline{L_n}$ to t in the graph $\mathcal{T}^{\mathcal{A}}$. Suppose there exists a $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$ with $\|\mathbb{T}\| < c$. Recall that $\|\mathbb{T}\|$ is the sum of the costs of all the nodes in $\mathbb{T}$. Thus, it suffices to show that the cost of the root of $\mathbb{T}$, plus the costs of all nodes rooted in the left and right subtrees of the root of $\mathbb{T}$, must be at least c. Our proof then proceeds inductively, working up from the leaves of $\mathbb{T}$. Let $[X_1, t, \omega_1], [X_2, t, \omega_1], \dots, [X_k, t, \omega_k]$ be the leaves of $\mathbb{T}$. Since each of the $[X_i, t, \omega_i]$'s are leaves, each of these paths must use only the linear edges from the original linear grammar edit distance graph $\mathcal{T}$ – so these edges must also exist in $\mathcal{T}^{\mathcal{A}}$. Thus for $1 \le i \le k$, the shortest path from X_i to t in $\mathcal{T}^{\mathcal{A}}$ is at most ω_i.

Now let $[A_\star^{p,q}, B_\star^{r,s}, \omega_\star]$ be any non-leaf node in $\mathbb{T}$, with left and right children $[A_L^{r,\ell}, \cdot, \omega_L]$ and $[A_R^{\ell+1,s}, \cdot, \omega_R]$ respectively. Let $\overline{\omega_L}$ and $\overline{\omega_R}$ be the sum of the costs of all nodes in the subtree rooted at $[A_L^{r,\ell}, \cdot, \omega_L]$ and $[A_R^{\ell+1,s}, \cdot, \omega_R]$, respectively. Note that because any node is included in the subtree rooted at itself, we include ω_L in the value $\overline{\omega_L}$ and ω_R is in the value $\overline{\omega_R}$.

Now suppose that the weight of the shortest path from $A_R^{\ell+1,s}$ to t and from $A_L^{r,\ell}$ to t in $\mathcal{T}^{\mathcal{A}}$ is at most $\overline{\omega_R}$ and $\overline{\omega_L}$ respectively. We would like to show that the shortest path from $A_\star^{p,q}$ to t in $\mathcal{T}^{\mathcal{A}}$ is at most $\omega_\star + \overline{\omega_R} + \overline{\omega_L}$.

Now since $[A_\star^{p,q}, B_\star^{r,s}, \omega_\star]$ ends in cloud (r,s), by property 1 of production-edit trees in $\mathcal{D}_{\mathcal{A}}$, it must be the case that $\mathcal{A}$ computes non-linear edges for the cloud (r,s). From $B_\star^{r,s}$, a non-linear edge e, corresponding to the production $B_\star \to A_L A_R$, is taken with splitting point ℓ. By property 2 of trees in $\mathcal{D}_{\mathcal{A}}$, the splitting point ℓ must have been considered by $\mathcal{A}$ when computing the cost of this edge. Thus, the cost of the edge e in $\mathcal{T}^{\mathcal{A}}$ is at most the cost of the shortest path from

A_L to t plus the cost of the shortest path from A_R to t. By the inductive hypothesis, the cost of e is then at most $\overline{\omega_R} + \overline{\omega_L}$. Since $\omega_\star$ is the cost of a path of consisting only of linear edges from $A_\star^{p,q}$ to $B_\star^{r,s}$, this path must also exists in $\mathcal{T}^{\mathcal{A}}$. Thus following this path from $A_\star^{p,q}$ to $B_\star^{r,s}$ and then taking e results in a path that exists in $\mathcal{T}^{\mathcal{A}}$, going from $A_\star^{p,q}$ to t, with cost at most $\omega_\star + \overline{\omega_R} + \overline{\omega_L}$, which is the desired result.

Finally, note that since $\mathcal{A}$ creates all *null* edges of the graph created by the exact algorithm $\mathcal{T}^{OPT}$, the above result holds in the case where one of the left or right children of $[A_\star^{p,q}, B_\star^{r,s}, \omega_\star]$ is a nullified node, since then the cost of the *null* edge is just the cost of nullifying the specified nonterminal, and the inductive hypothesis holds for the other, non-nullified, child. This completes all cases. Using this argument inductively, it follows that c must be no greater than the cost of the root of $\mathbb{T}$ plus the costs of all nodes rooted in the left and right subtrees of the root of $\mathbb{T}$, a contradiction, which completes the proof.

□

7.5 Proof of Theorem 8

Here we state a slightly more detailed statement for Theorem 8 and prove it.

Theorem (8)**.** *For any approximation algorithm $\mathcal{A} \in \mathcal{F}$ with precision functions α, β, there exists a tree $\mathbb{T} \in \mathcal{D}_{\mathcal{A}}$ and a Production-Edit Tree Mapping $\psi : V(\mathbb{T}_{OPT}) \to V(\mathbb{T})$ such that the following holds:*

If $\psi([X_i, Y_i, c_i])$ is not a leaf, we have

$$c'_i \leq c_i + (|p'_i - p_i| + |q'_i - q_i|) - (|r'_i - r_i| + |s'_i - s_i|) + 2\alpha(r_i, s_i) \qquad \text{(Non-leaf)}$$

If $\psi([X_i, Y_i, c_i])$ is a leaf, then

$$c'_i \leq \overline{c_i} + |p'_i - p_i| + |q'_i - q_i| + \beta(r_{i-1}, s_{i-1}) \qquad \text{(Leaf)}$$

Furthermore, if the children of $\psi([X_{i-1}, Y_{i-1}, c_{i-1}])$ were constructed in Case 1, then we have:

$$(|p'_L - p_L| + |q'_L - q_L|) + (|p'_R - p_R| + |q'_R - q_R|) \leq |r'_{i-1} - r_{i-1}| + |s'_{i-1} - s_{i-1}| + 2\beta(r_{i-1}, s_{i-1}) \quad (*)$$

Otherwise, the children of $\psi([X_{i-1}, Y_{i-1}, c_{i-1}])$ were created in Case 2. Then if both children are leaves of $\mathbb{T}$, then one of the following two inequalities is an upper bound for $c'_L + c'_R$:

$$\leq \overline{c_L} + c_R + |r'_{i-1} - r_{i-1}| + |s'_{i-1} - s_{i-1}| + 2\beta(r_{i-1}, s_{i-1}) \qquad (**)$$

$$\leq c_L + \overline{c_R} + |r'_{i-1} - r_{i-1}| + |s'_{i-1} - s_{i-1}| + 2\beta(r_{i-1}, s_{i-1}) \qquad (**)$$

Otherwise one of the next two inequalities is an upper bound for $c'_L + c'_R$:

$$\leq \overline{c_L} + c_R + |r'_{i-1} - r_{i-1}| + |s'_{i-1} - s_{i-1}| + 2\beta(r_{i-1}, s_{i-1}) - (|r'_R - r_R| + |s'_R - s_R|) + 2\alpha(r_R, s_R) \quad (***)$$

$$\leq c_L + \overline{c_R} + |r'_{i-1} - r_{i-1}| + |s'_{i-1} - s_{i-1}| + 2\beta(r_{i-1}, s_{i-1}) - (|r'_L - r_L| + |s'_L - s_L|) + 2\alpha(r_L, s_L) \quad (***)$$

Proof. We define ψ by explicitly constructing the nodes of the tree $\mathbb{T}$. We do this by constructing nodes to map to from the left and right children of any node at depth i in $\mathbb{T}_{OPT}$, and then inducting on i. For each non-leaf node $v \in \mathbb{T}_{OPT}$ for which we have already constructed a corresponding node $\psi(v) \in \mathbb{T}_{OPT}$, we *simultaneously* construct the nodes $\psi(v_R), \psi(v_L)$ corresponding to the right and left children v_R, v_L of v respectively. This occurs since the construction of $\psi(v_R)$ will depend on the splitting point used for the construction of $\psi(v_L)$, and vice-versa. Our mapping will be well-behaved, in the sense that the resulting tree $\mathbb{T}$ will be a valid production-edit tree in $\mathcal{D}_{\mathcal{A}}$. We proceed in cases. For each, we will bound the distance between the starting clouds with the goal of demonstrating $(*)$, and then apply Lemma 1 to obtain paths with costs that satisfy either the (Leaf) or (Non-leaf) property. Note that Lemma 1 guarantees that each time we use it to construct a path $[\cdot, B^{r'_i, s'_i}, c'_i]$ from $[\cdot, B^{r_i, s_i}, c_i]$, we will always have that $r_i \leq r'_i \leq s'_i \leq s_i$.

Building the root: Suppose the root of $\mathbb{T}_{OPT}$ is $[S^{1,n}, B^{r_1,s_1}, c_1]$ for some $B \in Q$. We begin by mapping the root $[S^{1,n}, B^{r_1,s_1}, c_1]$ of $\mathbb{T}_{OPT}$ to the root of $\mathbb{T}$. This is a special case, since the root is neither a left nor a right child, so we only need to satisfy the (Non-leaf) property (if the root is a leaf, only linear productions are made and our approximation will be optimal). By Lemma 1, we can construct a path $S^{1,n}$ to $B^{r'_1,s'_1}$, of weight at most $c_1 + 2\alpha(r_1, s_1)$ such that $\mathcal{A}$ constructs nonlinear edges in (r'_1, s'_1). We thus create the root node $[S^{1,n}, B^{r'_1,s'_1}, c'_1]$ in $\mathbb{T}$ corresponding to this path with $c'_1 \le c_1 + 2\alpha(r_1, s_1)$, and map only the root of $\mathbb{T}_{OPT}$ to it. Then the root satisfies (Non-leaf). Now from $B^{r'_1,s'_1}$ we can take the edge corresponding to the same non-linear production as the edge taken by OPT from B^{r_1,s_1}, allowing us to map a child of $[S^{1,n}, B^{r_1,s_1}, c_1]$ to a child of $[S^{1,n}, B^{r'_1,s'_1}, c'_1]$ such that the paths corresponding to both children begin at the same nonterminal.

Building the rest of the tree: Now suppose that we have defined ψ on all nodes up to depth $i-1$ in $\mathbb{T}_{OPT}$ by constructing the nodes they map to in $\mathbb{T}$ which satisfy the desired properties, and such that ψ does not violate any of the conditions of a production-edit tree mapping. For $N, M \in Q$, let $[N^{p_{i-1},q_{i-1}}, M^{r_{i-1},s_{i-1}}, c_{i-1}]$ be any node at depth $i-1$, and let $\psi([N^{p_{i-1},q_{i-1}}, M^{r_{i-1},s_{i-1}}, c_{i-1}]) = [N^{p'_{i-1},q'_{i-1}}, M^{r'_{i-1},s'_{i-1}}, c'_{i-1}] \in \mathbb{T}$ be the node that it is mapped to. We show how to construct both right and left children of $[N^{p'_{i-1},q'_{i-1}}, M^{r'_{i-1},s'_{i-1}}, c'_{i-1}]$ that we can map the right and left children of $[N^{p_{i-1},q_{i-1}}, M^{r_{i-1},s_{i-1}}, c_{i-1}] \in \mathbb{T}_{OPT}$ to. We first consider the case of the left child. So let $[A^{r_{i-1},\ell}, B^{r_i,s_i}, c_i]$ be the left child of $[N^{p_{i-1},q_{i-1}}, M^{r_{i-1},s_{i-1}}, c_{i-1}]$, coming from a production $M \to AC$, where $A, B \in Q$, and ℓ is some splitting point satisfying $r_{i-1} \le \ell \le s_{i-1}$. We need to consider several cases.

Case 1L (left child): $r'_{i-1} \le \ell < s'_{i-1}$.

Whenever this is the case we can apply Lemma 1 to a starting cloud (r'_{i-1}, ℓ') in order to construct a path starting at (r'_{i-1}, ℓ') that satisfies the desired properties. Since $r_i \le r'_i \le s'_i \le s_i$, there exists a splitting point ℓ' that is computed by $\mathcal{A}$ such that $|\ell - \ell'| \le \beta(r_{i-1}, s_{i-1})$. Then we have the identity (L):

$$|r'_{i-1} - r_{i-1}| + |\ell - \ell'| \le |r'_{i-1} - r_{i-1}| + \beta(r_{i-1}, s_{i-1}) \tag{L}$$

We now apply Lemma 1 on inputs $[A^{r_{i-1},\ell}, B^{r_i,s_i}, c_i]$ and (r'_{i-1}, ℓ'), giving us a path $[A^{r'_{i-1},\ell'}, B^{r'_i,s'_i}, c'_i]$, such that $c'_i \le c_i + (|r'_{i-1} - r_{i-1}| + |\ell' - \ell|) - (|r'_i - r_i| + |s'_i - s_i|) + 2\alpha(r_i, s_i)$. Then this node satisfies the (Non-leaf) property, and we set $\psi([A^{r_{i-1},\ell}, B^{r_i,s_i}, c_i]) = [A^{r'_{i-1},\ell'}, B^{r'_i,s'_i}, c'_i]$, completing the construction of the left child. The figure below illustrates the relationship between all four nodes in question (both parent-child pairs). Here one can see the identity (L) visually: the difference $\delta = |r'_{i-1} - r_{i-1}| + |\ell - \ell'|$ between the starting clouds of the two left children is at most $|r'_{i-1} - r_{i-1}| + \beta(r_{i-1}, s_{i-1})$.

Case 1R (right child): $r'_{i-1} \le \ell < s'_{i-1}$.

We now construct the right child of the same node $[N^{p'_{i-1},q'_{i-1}}, M^{r'_{i-1},s'_{i-1}}, c'_{i-1}] \in \mathbb{T}$ as before, using the same splitting point ℓ' as was used in Case $1L$. Note that the same splitting point **must** be used in both the construction of the right and left children of $[N^{p'_{i-1},q'_{i-1}}, M^{r'_{i-1},s'_{i-1}}, c'_{i-1}]$, otherwise the resulting tree $\mathbb{T}$ would not be a valid production-edit tree. Let $[C^{\ell+1,s_{i-1}}, D^{r_i,s_i}, c_i]$ be the right child of $[N^{p_{i-1},q_{i-1}}, M^{r_{i-1},s_{i-1}}, c_{i-1}]$. Note that while we are using the same notation r_i, s_i to denote the ending cloud of the right child that we used before for the left, the ending clouds of the right and left children will necessarily be distinct, and we use the same notation only to avoid introducing unnecessary variables. Then again, since $|(\ell'+1) - (\ell+1)| \le \beta(r_{i-1}, s_{i-1})$, we have our second identity (R):

$$|(\ell'+1) - (\ell+1)| + |s'_{i-1} - s_{i-1}| \le |s'_{i-1} - s_{i-1}| + \beta(r_{i-1}, s_{i-1}) \tag{R}$$

We now apply Lemma 1 on inputs $[C^{\ell+1,s_{i-1}}, D^{r_i,s_i}, c_i]$ and starting cloud $(\ell'+1, s'_{i-1})$, giving us a new path $[C^{\ell'+1,s'_{i-1}}, D^{r'_i,s'_i}, c'_i]$ which satisfies the (Non-leaf) property for the right child. We then set $\psi([C^{\ell+1,s_{i-1}}, D^{r_i,s_i}, c_i]) = [C^{\ell'+1,s'_{i-1}}, D^{r'_i,s'_i}, c'_i]$, completing the construction of the right child.

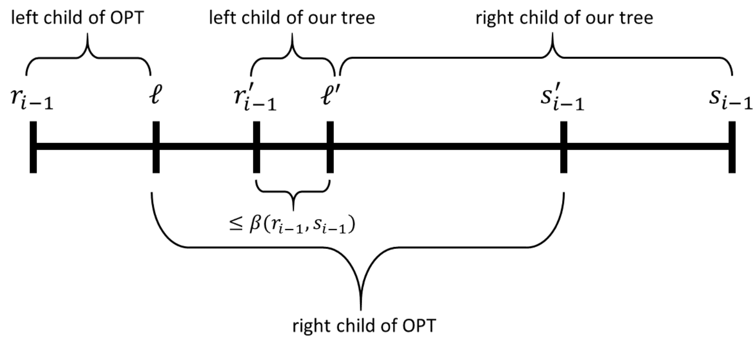
Now (L) is a bound on the distance between the starting clouds of the left child of $\mathbb{T}_{OPT}$ and the node it is mapped to by ψ, and (R) is the corresponding bound on the distance between the starting clouds of the right child and the node that it is mapped to by ψ. Adding the bounds $(L)+(R)$ produces the desired property $(*)$. Note that because equations (L) and (R) depend only on the starting cloud of the children, $(L)+(R)$ holds regardless of whether or not either child is a leaf node, since that is dictated by the ending cloud of a node. Thus we have satisfied the desired property $(*)$ in the case that $r'_{i-1} \leq \ell < s'_{i-1}$.

Case 1 Leaf: Now if the left child of $[N^{p_{i-1},q_{i-1}}, M^{r_{i-1},s_{i-1}}, c_{i-1}]$ is a leaf $[A^{r_{i-1},\ell}, t, c_i]$, then we similarly use Lemma 1 to map it to the leaf $[A^{r'_{i-1},\ell'}, t, c'_i]$ where the bound $c'_i \leq c_i + (|r'_{i-1} - r_{i-1}| + |\ell' - \ell|)$ holds, which satisfies the (Leaf) property. The same argument applies for when the right child is a leaf. Thus the above procedure stated in cases $(1L)$ and $(1R)$ to construct right and left children that satisfy the (Non-leaf) property also works to construct leaf children that satisfy the (Leaf) property. Thus we have satisfied the (Leaf) and (Non-leaf) properties in the case that $r'_{i-1} \leq \ell < s'_{i-1}$.

Case 2a: $\ell < r'_{i-1}$.

The difficulty in this case is that the substring (r_{i-1}, ℓ) which OPT derives in its left child is disjoint from the substring (r'_{i-1}, s'_{i-1}) which must be derived by the children of $[N^{p'_{i-1},q'_{i-1}}, M^{r'_{i-1},s'_{i-1}}, c'_{i-1}] \in \mathbb{T}$, thus we cannot utilize Lemma 1 to construct the left child of $[N^{p'_{i-1},q'_{i-1}}, M^{r'_{i-1},s'_{i-1}}, c'_{i-1}]$.

Case 2a – constructing the left child: First, fix ℓ' such that $|\ell' - r'_{i-1}| \leq \beta(r_{i-1}, s_{i-1})$ and such that the splitting point ℓ' is considered by $\mathcal{A}$. Note that we can always find such a ℓ' by the property of $\mathcal{A}$ and by definition of β. We will set the left child to be $[A^{r'_{i-1},\ell'}, \cdot, c'_i]$, which will be a leaf in $\mathbb{T}$. We leave the second argument blank, since we only need to bound the cost c'_i of deriving the string $x(r'_{i-1} : \ell')$ from A and then nullifying the remaining nonterminals, whether with linear or non-linear productions. Since we will need to closely consider the distance between several pairs of values, the relationships between the parameters in question are shown visually below:



Now since we can derive $x(r'_{i-1} : \ell')$ with at most $\beta(r_{i-1}, s_{i-1})$ insertions, we just need to bound the cost $null(A)$. Recall that $null(A)$ is the length of the shortest word derivable from A. We know that the right child $[A^{r_{i-1},\ell}, \cdot, c_i]$ in $\mathbb{T}_{OPT}$ derives a string of length $(\ell - r_{i-1})$ starting from A with cost at most $\overline{c_i}$, where $\overline{c_i}$ is the sum of the costs of all nodes in the subtree rooted at $[A^{r_{i-1},\ell}, \cdot, c_i]$. In the worst case this cost comes entirely from deletion edges, and thus

$null(A) \leq \overline{c_i} + (\ell - r_{i-1})$. Along with the $\beta(r_{i-1}, s_{i-1})$ insertions, we conclude that the cost of the left child satisfies $c_i' \leq \overline{c_i} + (\ell - r_{i-1}) + \beta(r_{i-1}, s_{i-1}) \leq \overline{c_i} + |r_{i-1}' - r_{i-1}| + \beta(r_{i-1}, s_{i-1}) \leq \overline{c_i} + |r_{i-1}' - r_{i-1}| + |\ell' - \ell| + \beta(r_{i-1}, s_{i-1})$, which satisfies (Leaf) (since (r_{i-1}, ℓ') is the starting cloud in question) . We set $\psi([A^{r_{i-1},\ell}, \cdot, c_i]) = [A^{r_{i-1}',\ell'}, \cdot, c_i']$, and map all children in the subtree rooted at $[A^{r_{i-1},\ell}, \cdot, c_i]$ to $[A^{r_{i-1}',\ell'}, \cdot, c_i']$. Note that *this is the only case where we map multiple nodes to the same place.*

Case 2a – constructing the right child: We now show how to construct the right child. Let $[C^{\ell+1,s_{i-1}}, D^{r_i,s_i}, c_i]$ be the right child of $[N^{p_{i-1},q_{i-1}}, M^{r_{i-1},s_{i-1}}, c_{i-1}]$ in $\mathbb{T}_{OPT}$. We fix the same ℓ' as before, since the splitting points of the right and left children must agree. We now have $\ell + 1 \leq \ell' + 1 \leq s_{i-1}' \leq s_{i-1}$, thus we can apply Lemma 1 to $[C^{\ell+1,s_{i-1}}, D^{r_i,s_i}, c_i]$ and the cloud $(\ell'+1, s_{i-1}')$ to map to a node $[C^{\ell'+1,s_{i-1}'}, D^{r_i',s_i'}, c_i']$, such that satisfies (Non-leaf) (or (Leaf) if the right child of $[N^{p_{i-1},q_{i-1}}, M^{r_{i-1},s_{i-1}}, c_{i-1}]$ is a leaf). Thus this is the desired right child. Now note that using Lemma 1, recalling as well the non-degenerate property of $\mathcal{A}$, we have two cases: if the right child is a leaf, we have the bound:

$$c_R' = c_i' \leq c_i + (s_{i-1} - s_{i-1}') + (\ell' + 1 - (\ell + 1))$$

$$\leq c_i + (s_{i-1} - s_{i-1}') + (r_{i-1}' - \ell) + \beta(r_{i-1}, s_{i-1})$$

Recalling from earlier that the left child had cost c_L' such that $c_L' \leq \overline{c_L} + (\ell - r_{i-1}) + \beta(r_{i-1}, s_{i-1})$, summing this bound with the bound for c_R' given above, we see that the that the sum of the costs of both children satisfies one of the first of the desired inequalities $(**)$.

Now if the right child is not a leaf, we have:

$$c_i' \leq c_i + (s_{i-1} - s_{i-1}') + (\ell' + 1 - (\ell + 1)) - (|r_i' - r_i| + |s_i' - s_i|) + 2\alpha(r_i, s_i)$$

$$\leq c_i + (s_{i-1} - s_{i-1}') + (r_{i-1}' - \ell) + \beta(r_{i-1}, s_{i-1}) - (|r_i' - r_i| + |s_i' - s_i|) + 2\alpha(r_i, s_i)$$

Summing the above inequality with the same bound on c_L' as for the leaf case gives the first of the desired inequalities $(***)$.

Case 2b : $\ell \geq s_{i-1}'$.

This case is entirely symmetric to case 2a. In this case the substring $(\ell+1, s_{i-1})$ which OPT derives in its right child is again disjoint from the substring (r_{i-1}', s_{i-1}') which must be derived by the children of $[N^{p_{i-1}',q_{i-1}'}, M^{r_{i-1}',s_{i-1}'}, c_{i-1}'] \in \mathbb{T}$. Thus the same procedure as in case 2 works, except we instead nullify the starting non-terminal C of the right child in $\mathbb{T}$, and then apply Lemma 1 to construct the left child, such that both satisfies all desired properties. Here, the children will satisfy the second inequality of either $(**)$ or $(***)$, instead of the first as in Case 2*a*. This completes all cases.

□

Remark 2. *Note that in the case where $\ell < r_{i-1}'$, or $\ell \geq s_{i-1}'$, and we cannot apply Lemma 1 to construct the left child, and we instead construct a node $[A^{r_{i-1}',\ell'}, \cdot, c_i']$, we are bounding c_i' as the total cost of inserting the entire substring $x(r_{i-1}' : \ell')$ from A and then nullifying A. Now the cost of nullifying A may involve making non-linear productions and nullifying the resulting nonterminals, so in actuality $[A^{r_{i-1}',\ell'}, \cdot, c_i']$ may not be a leaf of a valid production-edit tree, it may have nullified children nodes. However, the bound we place on c_i' is an upper bound on the cost of deriving $x(r_{i-1}' : \ell')$ from A in the approximation graph $\mathcal{T}^{\mathcal{A}}$, and thus is a bound on the total cost of all the nodes that could be in the subtree rooted at $[A^{r_{i-1}',\ell'}, \cdot, c_i']$. Therefore, the bound on c_i' does indeed satisfy the (Leaf) property, and so we can consider $[A^{r_{i-1}',\ell'}, \cdot, c_i']$ to be a leaf of $\mathbb{T}$*

Lemma 2. *Any k-ultralinear grammar can be converted into a k^*-ultralinear language in the above normal form, where $k^* \leq k\log(p)$, and p is the maximum number of nonterminals on the right hand side of any production.*

Proof. Consider any production $A \to A_1A_2\dots A_m$, where $A \in Q_t$ and A_j's are in partitions of lower index for $1 \leq j \leq m$. We add $\log(p)$ new partitions between Q_t and Q_{t-1}. We then make new nonterminals $A_1^1, A_2^1, \dots A_{\lceil m/2\rceil}^1$, and set the only production of each to be $A_i^1 \to A_{2i-1}A_{2i}$ for $i = 1, 2, ..., \lceil m/2\rceil$. If m is odd than $A_{\lceil m/2\rceil}^1 \to A_m$ will be the only production of the last nonterminal. We then repeat the process, creating nonterminals $A_1^2, \dots A_{\lceil m/4\rceil}^2$ and setting $A_i^2 \to A_{2i-1}^1A_{2i}^1$. Finally, we create the production $A \to A_1^{\lceil\log(m)\rceil}A_2^{\lceil\log(m)\rceil}$. We place the terminals A_j^i in the partition that is depth $\lceil\log(m)\rceil - 1 + i$ lower than Q_t. Furthermore, for any production $A \to B$ where $A \in Q_t$ and $B \in Q_l$ with $l < t$, we can add a new non-termianl $B' \in Q_l$ such that its only production is $B' \to \epsilon$. We then change the production to $A \to BB'$. This does not increase the number of partitions. Doing the first process for all productions, the resulting grammar has at most $k\log(p)$ partitions, and after both processes at most $p|P|$ new nonterminals, since every production has at most p nonterminals on the right hand side. □

References

[1] A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18 – 31, 1980.

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

[5] Nicola Conci Andrea Rosani and Francesco G. De Natale. Human behavior recognition using a context-free grammar. *Journal of Electronic Imaging*, 2014.

[6] 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.

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

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

[9] Dan Gusfield Balaji Venkatachalam and Yelena Frid. Faster algorithms for rna-folding using the four-russians method. 2013.

[10] L. Balke and K.H. Böhling. Einführung in die automatentheorie und theorie formaler sprachen. *BI Wissenschaftsverlag*, 1993.

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

[12] Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia V. Williams. Truly sub-cubic algorithms for language edit distance and rna folding via fast bounded-difference min-plus product. FOCS, 2016.

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

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

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

[16] S. Ginsburg and E.H. Spanier. Finite-turn pushdown automata. *SIAM J. Comput.*, (4), 1966.

[17] Parikshit Gopalan, TS Jayram, Robert Krauthgamer, and Ravi Kumar. Estimating the sortedness of a data stream. SODA, pages 318–327, 2007.

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

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

[20] 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.

[21] Yijie Han and Tadao Takaoka. An $o(n^3 \log\log n/\log^2 n)$ time algorithm for all pairs shortest paths. In *SWAT*, pages 131–141, 2012.

[22] John E Hopcroft and Jeffrey D Ullman. Formal languages and their relation to automata. 1969.

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

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

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

[26] Mark Johnson. Pcfgs, topic models, adaptor grammars and learning topical collocations and the structure of proper names. *ACL*, 2010.

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

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

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

[30] Donald E Knuth. A generalization of dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5, 1977.

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

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

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

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

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

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

[37] Darnell Moore and Irfan Essa. Recognizing multitasked activities from video using stochastic contextfree grammar. 2002.

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

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

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

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

[42] Andrea Rosani, Nicola Conci, and Francesco G. De Natale. Human behavior recognition using a context-free grammar. *Journal of Electronic Imaging.*

[43] Z Shang Y Du R.R Gutell, J.J. Cannone and M.J Serra. A story: unpaired adenosine bases in ribosomal rnas. *Journal of Mol Biology*, 2010.

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

[45] 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.

[46] Michael Saks and C Seshadhri. Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1698–1709. SIAM, 2013.

[47] 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.

[48] Leslie G. Valiant. The equivalence problem for deterministic finite-turn pushdown automata. *Information and Control*, 25, 1974.

[49] Balaji Venkatachalam, Dan Gusfield, and Yelena Frid. Faster algorithms for RNA-folding using the four-russians method. WABI, 2013.

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

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

[52] Ryan Williams. A new algorithm for optimal constraint satisfaction and its implications. In *International Colloquium on Automata, Languages, and Programming*, pages 1227–1237, 2004.

[53] Glynn Winskel. *The formal semantics of programming languages: an introduction.* 1993.

[54] Gerhard J. Woegingers. Space and time complexity of exact algorithms: Some open problems. *Parameterized and Exact Computation*, 3162, 2004.

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

[56] 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.