Fast & Space-Efficient Approximations of Language Edit Distance and RNA folding: An Amnesic Dynamic Programming Approach

Barna Saha
College of Information and Computer Science
University of Massachusetts Amherst
Amherst, MA
barna@cs.umass.edu

***Abstract*—Dynamic programming is a basic, and one of the most systematic techniques for developing polynomial time algorithms with overwhelming applications. However, it often suffers from having high running time and space complexity due to (a) maintaining a table of solutions for a large number of sub-instances, and (b) combining/comparing these solutions to successively solve larger sub-instances. In this paper, we consider a canonical cubic time and quadratic space dynamic programming, and show how improvements in both its time and space uses are possible. As a result, we obtain fast small-space approximation algorithms for the fundamental problems of *context free grammar recognition* (the basic computer science problem of parsing), *the language edit distance* (a significant generalization of string edit distance and parsing), and *RNA folding* (a classical problem in bioinformatics). For these problems, ours are the first algorithms that break the cubic-time barrier of any combinatorial algorithm, and quadratic-space barrier of "any" algorithm significantly improving upon their long-standing space and time complexities. Our technique applies to many other problems as well including string edit distance computation, and finding longest increasing subsequence.**

Our improvements come from directly grinding the dynamic programming and looking through the lens of language edit distance which generalizes both context free grammar recognition, and RNA folding. From known conditional lower bound results, neither of these problems can have an exact combinatorial algorithm (one that does not use fast matrix multiplication) running in truly subcubic time. Moreover, for language edit distance such an algorithm cannot exist even when nontrivial multiplicative approximation is allowed. We overcome this hurdle by designing an additive-approximation algorithm that for any parameter $k > 0$, uses $O(nk \log n)$ space and $O(n^2 k \log n)$ time and provides an additive $O(\frac{n}{k} \log n)$-approximation. In particular, in $\tilde{O}(n)$[1] space and $\tilde{O}(n^2)$ time it can solve deterministically whether a string belongs to a context free grammar, or ϵ-far from it for any constant $\epsilon > 0$. We also improve the above results to obtain an algorithm that outputs an $\epsilon \cdot n$-additive approximation to the above problems with space complexity $O(n^{2/3} \log n)$. The space complexity remains sublinear in n, as long as $\epsilon = o(n^{-\frac{1}{4}})$. Moreover, we provide the first MapReduce and streaming algorithms for them with multiple passes and sublinear space complexity.

I. Introduction

Dynamic programming (DP) is one of the most systematic approaches for developing exact polynomial time algorithms. It implements a recursive procedure, but stores the result from each computed subproblem in a table that can be accessed over and over, 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. Therefore, DP based approaches often have high degree of polynomial time complexity, and have high space requirements. Many techniques such as the *Four-Russians method* have been developed in the literature to make dynamic programming fast [23], [36], [53]. Unfortunately, their speed-up gains have mostly been restricted to only poly-logarithmic factors. On the other hand, if we allow for an approximate answer, a major improvement in running time as well as space usage may be possible.

With this motivation, in this paper we explore a canonical cubic-time and quadratic-space DP, and show how major improvements both in space and time are possible if approximation is allowed by reducing the number of subinstances for which a solution must be stored, and by reducing the number of subinstances that must be consulted to solve larger instances. This leads to new or improved results for many fundamental problems including *context free grammar recognition*, *the language edit distance*, and *RNA folding*.

Context Free Grammar Recognition & Language Edit Distance. Context free grammar recognition is a basic computer science question that given a context free grammar (CFG) G and a string σ of length n over alphabet Σ, solves the core parsing problem, that is it determines whether σ belongs to the language $\mathcal{L}(G)$ generated by G. The canonical Cocke-Younger-Kasami (CYK) algorithm (developed in 1960's) and

[1] $\tilde{O}(n)$ implies $O(n \text{poly} \log n)$.

Earley's parser (developed in 1968-70) [24] use dynamic programming that runs in $O(n^3)$ time and requires $O(n^2)$ space. The first improvement over these $O(n^3)$ running time came in the breakthrough result of L. Valiant. In his dissertation thesis, Valiant showed how to use many fast boolean matrix multiplications to design a CFG recognition algorithm in $O(n^{\omega})$ time [51], where $\omega = 2.373$ is the fast matrix multiplication exponent [19], [52]. Despite its vast theoretical significance, use of fast matrix multiplication limits the practicality of Valiant's approach, and is mostly outperformed by cubic time "combinatorial" algorithms based on dynamic programming. While the pursuit for efficient parsing algorithms continued to grow for various special classes of grammars [7], [18], [43], [45], two remarkable conditional lower bound results, one by Lee [34], and the other, a relatively recent one, by Abboud, Backurs and V. Williams [1] show that for general context free grammars, neither it is possible to improve beyond Valiant's algorithm, nor it is possible to develop any combinatorial algorithm for CFG recognition that runs in truly subcubic time

A problem that significantly generalizes CFG recognition is *the language edit distance* (LED) [2], [38]. Given a grammar G and a string σ of length n over an alphabet Σ, the language edit distance finds the minimum number of insertions, deletions, and substitutions required to convert σ into a member of $\mathcal{L}(G)$. That is, while CFG recognition checks for membership, LED finds the actual distance. It also significantly generalizes the string edit distance computation. Introduced by Aho and Peterson in 1972 [2], for over forty years an $O(n^3)$ dynamic programming algorithm for LED had the best known running time. Relatively recently, an $(1+\epsilon)$-multiplicative approximation algorithm for LED has been developed that runs in $\tilde{O}(\frac{n^{\omega}}{\epsilon^4})$ time [48]. It is only in the last year that an exact subcubic algorithm for LED has been found with $O(n^{2.8244})$ running time [13]. A fast combinatorial algorithm for LED will have huge impact due to its vast number of applications in multiple domains [21], [22], [26], [30], [37], [44], [46], [53], [54], as well as its close connection to many fundamental graph problems [48]. However, the algorithms of [13], [48] use fast matrix multiplication, and arguably so, since without fast matrix multiplication, the lower bound for parsing implies there cannot even exist any nontrivial multiplicative approximation algorithm for LED that runs in subcubic time.

In the absence of a subcubic combinatorial exact algorithm, it is natural to ask whether one can use approximation to design faster algorithms. The apparent inapproximability result of LED questions the viability of this approach. Similar in flavor, Alon, Krivelevich, Newman and Szegedy initiated the study of approximate membership checking of formal languages [4] where one wants to distinguish fast if $\sigma \in \mathcal{L}(G)$, or ϵ-far from it (that is the language edit distance of σ with respect to G is at least ϵn) . While efficient testers are known for regular languages [4], the complex structure of context free grammars make designing a tester for CFG membership checking highly challenging. It is only for simple classes of grammars such as Dyck (the language of well-balanced parenthesis for which parsing time is linear), better testers are known [42]. In general, though very basic in appeal, it is unclear whether one can beat the parsing time to check approximate membership in languages. In this paper, we make significant progress towards this question. Using our recipe, the so called *amnesic dynamic programming*, we design the first combinatorial algorithms for these problems that beat the parsing time. This improvement came after more than four decades through developing additive approximation. Note that by a t-additive approximation, we mean a solution that is within $\pm t$ away from the optimal.

Theorem 1. *Given a parameter $k \geq 1$, there exists an algorithm which for any grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$ and $\sigma \in \Sigma^*$, $|\sigma| = n$, computes an $O(\frac{n}{k}\log n)$-additive approximation for LED in time $O(n^2 k \log n)$ and space $O(nk \log n)$.*

For example, in $\tilde{O}(n^{2.5})$ time we get a combinatorial algorithm for LED that estimates the distance within $\sqrt{n}$-additive error. By substituting $k = O(\frac{\log n}{\epsilon})$, we get the following direct corollary for an approximate CFG recognizer.

Corollary 1. *There exists an algorithm which for any grammar G and $\sigma \in \Sigma^*$, $|\sigma| = n$, can distinguish between whether $\sigma \in \mathcal{L}(G)$ or ϵ-far from it in $O(n^2 \log^2 n)$ time and $O(n \log^2 n)$ space for any constant $\epsilon > 0$.*

RNA Folding. We now describe another problem that is central in bioinformatics for which our algorithm also leads to significant improvement. RNA folding introduced by Nussinov and Jacobson in 1980 [40] is

[2]For simplicity of exposition, we will consider the grammar size $|G|$ to be constant, and will not explicitly mention the dependency of $|G|$ on space and time.

[3]Lee's reduction is from boolean matrix multiplication under the assumption that the dependency on grammar size is linear. Abboud et al.'s hardness result is based on the conjectured hardness of k-clique and works even for constant size grammars.

the following optimization problem. Let Σ be a set of letters and let $\Sigma' = \{c' \mid c \in \Sigma\}$ be the set of "matching" letters, such that for every letter $c \in \Sigma$ the pair c, c' matches. Given a sequence of n letters over $\Sigma \cup \Sigma'$, the RNA folding problem asks for the maximum number of non-crossing pairs $\{i, j\}$ such that the ith and jth letter in the sequence match. In particular, if letters in positions i and j are paired and if letters in positions k and l are paired, and $i < k$ then either they are nested, i.e., $i < k < l < j$ or they are non-intersecting, i.e., $i < j < k < l$. (In nature, there are 4 types of nucleotides in an RNA molecule, with matching pairs A, U and C, G, i.e., $|\Sigma| = 2$). There exists a simple $O(n^3)$ dynamic programming to obtain the optimal folding with polylogarithmic improvements [53], and a trivial linear-time approximation algorithm that selects the subsequence on a single letter which maximizes the matching to get an $\frac{1}{|\Sigma|}$-approximation. When fast matrix multiplication is allowed, [13] also implies an exact $O(n^{2.8244})$-time algorithm for RNA folding, and using the $(1+\epsilon)$-approximation for LED [48], an additive ϵn-approximation in $O(\frac{n^\omega}{\epsilon^4})$ time can be obtained.

We can rephrase RNA folding as follows. We are given a CFG with productions $S \rightarrow SS \mid \epsilon$ and $S \rightarrow xSx' \mid x'Sx$ for any $x \in \Sigma$ with matching $x' \in \Sigma'$. The goal is to find a maximum size subsequence of σ that is in $\mathcal{L}(G)$, that is we want to perform minimum number of *insertions* and *deletions* of symbols on a given string σ that will generate a string consistent with the above grammar. This is essentially a maximization version of LED where only insertions and deletions (and no substitutions) are allowed. Since, our Theorem 1 holds for just insertions and deletions, we not only get the first combinatorial algorithm with nontrivial approximation guarantees running in subcubic time, but also a significant improvement over [48] for RNA folding.

Corollary 2. *There exists an algorithm for RNA folding that gives an $O(\frac{n}{k}\log n)$ additive approximation for RNA sequences of length n, running in $O(n^2 k \log n)$ time and requiring $O(nk\log n)$ space.*

Space Complexity & Other Models of Computation: Note that the space requirements in Theorem 1, Corollary 1 and 2 are $O(nk\log n)$. Previously, for all known algorithms of LED, CFG recognizer, and RNA folding (except for the trivial $\frac{1}{|\Sigma|}$-approximation algorithm, and for the extreme special cases like string edit distance), irrespective of whether they were exact or approximate, and used fast matrix multiplication or not, the space requirement was $O(n^2)$ [2], [13], [38], [48], [51], [53]. This is a major bottleneck for large data. We provide the first approximation algorithm that breaks this barrier.

In fact, we show again by our recipe of amnesic dynamic programming, how to maintain an approximate DP solution in sublinear space. Therefore, the space complexity of DP is not a bottleneck for these problems. In particular, we prove the following theorem.

Theorem 2. *Given two parameters γ and q such that $\gamma > \sqrt{q} \geq 1$, there exist efficient algorithms for LED, RNA folding, and approximate CFG recognizer that use space $O(\max(\frac{n^2}{q}, \frac{\gamma^2 \log n}{\sqrt{q}}))$ and achieve an additive approximation of $O(\frac{n\sqrt{q}\log n}{\gamma})$.*

For any $\epsilon < 1$, setting $\gamma = \frac{\sqrt{q}\log n}{\epsilon}$, we get an ϵn-additive approximation for LED in space $\tilde{O}(\frac{n^{2/3}}{\epsilon^{4/3}})$ by setting $q = \epsilon^{4/3} n^{4/3}$. Therefore, as long as $\epsilon = o(n^{-1/4})$ (or the additive approximation is $n^{3/4+\delta}$ for some $\delta > 0$), the space requirement is sublinear. For the special case of linear grammars, this bound can be further improved to give sublinear space algorithm with additive error $n^{1/2+\delta}$. As shown in [25], string edit distance computation can be reduced to computing LED over a linear grammar of constant size. Thus our result also implies similar space vs approximation trade-offs for computing string edit distance.

Theorem 3. *Given two parameters γ and q such that $\gamma \geq \sqrt{q} \geq 1$, there exists an efficient algorithm for LED for linear grammar and string edit distance computation on constant alphabet size that uses space $O(\max(\frac{n^2}{q}, \gamma))$ and achieves an additive approximation of $O(\frac{n}{\gamma}\sqrt{q})$.*

We note that this saving in space comes at a cost of increasing time, and we do not yet know how to combine results of Theorem 1 and Theorem 2 to obtain simultaneously sublinear space and subcubic time.

MapReduce and Streaming. To get Theorem 2, we do not need to assume random access to the input string. In fact, using sublinear number of sequential passes suffice. We show with only $O(\epsilon^{8/3} n^{2/3})$ passes over the input, it is possible to obtain the same space and approximation bound. This gives a sublinear-pass streaming algorithm for LED, RNA folding and approximate CFG membership using sublinear space. Previously streaming algorithms were known only for recognition problem of Dyck languages [35] and when very simple edits were allowed on it [31].

Our algorithm can also be efficiently implemented in the popular MapReduce setting [27], [33], thereby

providing the first parallel algorithms for these problems. In particular, the algorithm requires $O(\epsilon^{4/3}n^{1/3})$ machines each with $\tilde{O}(\frac{n^{2/3}}{\epsilon^{4/3}})$ space, and $O(\epsilon^{4/3}n^{1/3})$ rounds. With $\tilde{O}(n^{1-\delta})$ space per machine, the number of rounds decreases to $O(n^{\delta})$ to maintain the same approximate solution.

Further Applications. Our algorithms are extremely simple and deterministic. We envision many further applications of this method. For example, using Theorem 1, we can get a single-pass *deterministic* streaming algorithm for the longest increasing subsequence (LIS) (and hence for distance to monotonicty: $n - LIS$) with a half-page analysis of it which shows an ϵn-additive approximation for LIS can be maintained in $O(\frac{\log n}{\epsilon})$ space. The previously known algorithm of [49] was randomized, and had a space requirement of $O(\frac{\log^2 n}{\epsilon})$ for the same approximation factor. A deterministic algorithm was designed in [39] that obtains an $(1+\epsilon)$-multiplicative approximation algorithm for distance to monotonicity which also implies an ϵn-additive approximation but with $O(\frac{\log^2 n}{\epsilon^2})$ space.

Theorem 4. *There exists a deterministic algorithm for LIS that uses $O(\frac{\log n}{\epsilon})$ space and computes an ϵn-additive approximation of LIS.*

From our multi-pass streaming algorithm using Theorem 2, a simple one-pass streaming algorithm for edit distance in the asymmetric setting falls off directly [5], [49]. In the asymmetric streaming model, we have full random access to one of the strings and streaming access to the other. Here the bounds are slightly suboptimal compared to [49], though we believe the analysis can be made tighter for this special case.

Theorem 5. *There exists a deterministic one pass streaming algorithm for computing string edit distance in the asymmetric setting that uses $O(\frac{\sqrt{n}}{\epsilon})$ space and returns an ϵn-additive approximation of the string edit distance.*

We believe our technique will find wide-applicability to directly improve dynamic programming algorithms either in time, or in space, or both. As immediate applications of our method, one can explore the large number of sequence problems considered in [16], the popular word-wrap problem, the classical optimal binary search tree problem [28] etc. Many high-dimensional problems which exhibit certain smoothness in their solution to subproblems are all amenable to our method. Since the algorithms are based on forgetting to maintain/use some of the states of a typical DP algorithm, we call this an amnesic dynamic programming approach.

A. Techniques.

A context-free grammar (grammar for short) is a 4-tuple $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$ where $\mathcal{N}$ and Σ are finite disjoint collection of nonterminals and terminals respectively. $\mathcal{P}$ is the set of productions of the form $A \to \alpha$ where $A \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*$. S is a distinguished symbol in $\mathcal{N}$ known as the *start* symbol.

For two strings $\alpha, \beta \in (\mathcal{N} \cup \Sigma)^*$, we say α directly derives β, written as $\alpha \Rightarrow \beta$, if one can write $\alpha = \alpha_1 A \alpha_2$ and $\beta = \alpha_1 \gamma \alpha_2$ such that $A \to \gamma \in \mathcal{P}$. Thus, β is a result of applying the production $A \to \gamma$ to α. $\mathcal{L}(G)$ is the context-free language generated by grammar G, i.e., $\mathcal{L}(G) = \{w \in \Sigma^* \mid S \stackrel{\star}{\Rightarrow} w\}$, where $\stackrel{\star}{\Rightarrow}$ implies that w can be derived from S using one or more production rules.

A grammar G is said to be in Chomsky Normal Form (CNF), if every production in $\mathcal{P}$ is of the form $A \to BC$ or $A \to x$ where $A, B, C \in \mathcal{N}$ and $x \in \Sigma$. In addition, CNF allows to have $S \to \varepsilon$ for producing empty string. It is well-known that every context free grammar can be represented in CNF. We thus consider our grammar G to be in Chomsky Normal Form, but when developing algorithms, we allow some additional types of productions such as we allow $A \to \varepsilon$ to be in $\mathcal{P}$ for any non-terminal $A \in \mathcal{N}$. We always assume $\varepsilon \in \mathcal{L}(G)$ to keep LED bounded.

The canonical CYK algorithm which [2] builds upon for LED creates a two-dimensional $n \times n$ DP table $\mathcal{T}$, where $\mathcal{T}(i,j)$, $i \leq j$, stores parsing information for the substring $\sigma_i\sigma_{i+1}...\sigma_j$, abbreviated as $\sigma(i:j)$. To obtain $\mathcal{T}(i,j)$, it checks the parsing information for $\sigma(i:\ell)$, and $\sigma(\ell+1,j)$ for every $\ell \in [i, j-1]$. For example, if a non-terminal B derives $\sigma(i:\ell)$, and non-terminal C derives $\sigma(\ell+1,j)$, and there is a production $A \to BC \in \mathcal{P}$, then A can derive $\sigma(i:j)$. These intermediate points ℓ are referred to as "break-points". LED in addition maintains the cost of derivation, and adds them accordingly. This results in an $O(n^3)$ time algorithm as the DP fills an $O(n^2)$ sized table, and filling each table entry requires looking for $O(n)$ subproblems.

As a first step, we would like to reduce the complexity of DP by considering only a subset of entries according to some probability distribution. Here our main source of inspiration are the works of [49] and [20]. In [20], Gopalan, Jayram, Krauthgamer and Kumar use a time-varying synopsis of a dynamic programming table to estimate the number of inversions in a stream in "small" space for approximate distance to monotonicity and LIS. The approximation bound was later improved upon by Saks and Seshadhri to give any arbitrary ϵn additive

approximation [49]. Since, these are one-dimensional problems, that is they maintain a one-dimensional DP table, say $s[1], s[2]...., s[n]$, when computing $s[t]$, t can be viewed as time. The algorithms of [20], [49] use a complex probability distribution to sample from these entries that becomes sparse with time. It results in a randomized algorithm that improves the space complexity to $O(\log^2 n)$ when allowing additive approximation.

The problem that we have in hand is significantly more complex. First, the notion of time is not so clear when maintaining a two-dimensional DP table. Moreover both [20], [49] are tailored towards saving space, and use a complicated probability distribution that is hard to analyze for the complex problem of LED. They do not provide any improvement on running time than what was previously known for LIS [3], [17]. One may be tempted to sample according to substring length as DP computes the entries in that order to obtain a time-varying sequence. We analyze such schemes in [25] for special classes of grammars using their additional structures, but for general LED such sampling schemes provide no nontrivial guarantees.

Here we consider the following idea. When computing for $\mathcal{T}(i, j)$, we divide the interval $[i, j]$ into $O(k \log n)$ subintervals of almost geometrically increasing size maintaining certain subset property once from i, and once from j. Roughly speaking, we consider the subintervals $[i, i+k-1], [i+k, i+2k-1], [i+2k, i+4k-1]$ and so on. Similarly, we consider the subintervals $[j, j-k+1], [j-k, j-2k+1], [j-2k, j-4k+1]$ and so on. Then we start selecting points $\ell \in [i, j-1]$ such that if ℓ lies in a short subinterval it is selected/sampled more vigorously. For example, if ℓ is in a subinterval of length $2^t k$, then it is sampled with probability $\frac{1}{2^t}$. Thus the sampling becomes sparse in the middle. We compute all $\mathcal{T}(i, j)$, but instead of combining $\mathcal{T}(i, \ell)$ and $\mathcal{T}(\ell + 1, j)$ for all $\ell \in [i, j-1]$, we consider ℓ only if it is selected. Moreover this process of selecting a sample of break-points can be done deterministically. The intuition behind the algorithm is as follows. Suppose, an optimal algorithm always uses the break-point $i + \lfloor (i-j)/2 \rfloor$ when computing LED for $\sigma(i:j)$. Then, the corresponding parse tree will have $O(\log n)$ height. Thus even if we allow an additive error of $\lfloor (i-j)/k \rfloor$ while computing LED for $\sigma(i : j)$, we still get an overall $O(\frac{n}{k} \log n)$-additive approximation. However, if the optimal algorithm always uses the break-point $i+1$ when considering $\sigma(i : j)$, then we cannot allow our algorithm to have an additive error of $\lfloor (i-j)/k \rfloor$. We must follow the optimal algorithm very closely here by selecting more break-points. This leads to the desired improvement in running time for approximating LED, RNA folding and CFG recognition. Moreover, by treating $i = 0$, and only sampling from j, we also get a deterministic algorithm for LIS with $O(\log n)$ space complexity as opposed to the randomized $O(\log^2 n)$ algorithm of [49]. Moreover, we show at any time the DP needs to maintain only $O(nk \log n)$ entries to get an $O(\frac{n \log n}{k})$-additive approximation for LED, thus improving the space.

We would like to further reduce the space usage to sublinear in n. To do so, naturally, we would like to consider only a subset of entries from $\mathcal{T}$, and then use our approximate computation only for these selected entries. Here, we consider the simplest possible sampling scheme–choose $O(\frac{n^2}{q})$ entries randomly and uniformly for a given parameter q. The randomization is in fact not required, and the points can be chosen deterministically maintaining the desired property. When computing for $\mathcal{T}(i, j)$, where (i, j) is sampled, if we would like to combine $\mathcal{T}(i, \ell)$ and $\mathcal{T}(\ell + 1, j)$, but one or both of them are not selected, then we move to their nearest sampled points and use them instead. This simple recipe does not quite work by itself. We need to incorporate some on-the-fly computation to allow for dense sampling of break-points near i and j as in the time-efficient algorithm. In some more details, if the break-point ℓ is "close" to i (same for j) and $\mathcal{T}(i, \ell)$ is not selected, then we compute $\mathcal{T}(i, \ell)$ on the fly using our time-efficient algorithm. This gives our desired space bound of Theorem 2. Handling the propagation of error through recursive calls becomes quite intricate at this point. As mentioned, this also gives an efficient MapReduce and multi-pass streaming algorithms for the problems.

B. Further Related Work

For the special case of string edit distance computation, extensive work has been done that improves on the $O(n^2)$ running time of the dynamic programming and returns approximate solution [5], [6], [10], [11], [15], [32], [41]. When the optimum distance is small, interesting streaming algorithms have also been developed [12], [14]. For Dyck language edit distance problem, [47] gives a linear time multiplicative polylogarithmic approximation improving on the $O(n^3)$ running time of the dynamic programming. Again, when distance is small, exact algorithms with better running times can be obtained [9]. For general LED, the only known multiplicative approximation bound is from [48]. Prior to this work, there existed no combinatorial algorithm that beats the time or space complexity of the classical dynamic programming. Recently, in [25], we developed

faster combinatorial algorithms for special classes of context free grammars, specifically for the ultralinear grammars. However, they do not provide any nontrivial guarantees for arbitrary context free languages. Regarding lower bounds, Backurs and Indyk showed the exact computation of string edit distance is not possible in truly subquadratic time unless the strong exponential time hypothesis is false [8]. For RNA folding and LED, the best known lower bound is $\Omega(n^\omega)$ for exact computation [1]. Moreover, [1] rules out any combinatorial exact algorithm in truly subcubic time.

Note. Due to space limitation, this extended abstract only contains our result pertaining to Theorem 1. The remaining results and the missing proofs are available in the full version.

II. Preliminaries

Definition 1 (Language Edit Distance (Aho & Peterson'72 [2])). *Given a grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$ and $\sigma \in \Sigma^*$, the language edit distance between G and σ is defined as*

$$d_G(G, \sigma) = \min_{z:\in \mathcal{L}(G)} \mathrm{d}_{ed}(\sigma, z)$$

where d_{ed} is the standard edit distance (insertion, deletion and substitution) between σ and z. If this minimum is attained by considering $z \in \mathcal{L}(G)$, then z serves as a witness for $\mathrm{d}_G(G, \sigma)$.

A t-additive approximation for language edit distance with $t \geq 0$ returns an estimate $\hat{\mathrm{d}}_G(G, \sigma)$ such that $\mathrm{d}_G(G, \sigma) \leq \hat{\mathrm{d}}_G(G, \sigma) \leq \mathrm{d}_G(G, \sigma) + t$.

Dynamic Programming For Language Edit Distance (LED).: We now describe the dynamic programming algorithm from [2] for computing LED optimally in $O(n^3)$ time and $O(n^2)$ space. For simplicity of exposition, we consider grammar size, $|G| = |\mathcal{P}| + |\mathcal{N}| + |\Sigma|$ to be constant, and do not explicitly discuss the dependency on grammar size in runtime or space complexity.

The first step in the algorithm of [2] is to add error-producing rules to create an augmented grammar $G' = (\mathcal{N}', \Sigma, \mathcal{P}', S)$ that allow editing the input string by paying an appropriate cost. We use slightly simpler rules as in [13], [48] by violating CNF slightly. Let $s(Z \rightarrow \alpha)$ represent the score/cost for applying the production $Z \rightarrow \alpha$. The score of all original rules are 0. For every $a \in \Sigma$, we add a new non-terminal $X_a \rightarrow a$ with a score of $s(X_a \rightarrow a) = 0$.

Substitution. To allow for substitution of symbols, we add for every $a, b \in \Sigma, a \neq b$, $X_a^b \rightarrow b$ with a score of 1 with a new non-terminal X_a^b.

Insertion. To allow for insertion of symbols, we add the following rules with a new non-terminal I for every $a \in \Sigma$ and for every non-terminal $X \in \mathcal{P}$

$$I \rightarrow X_a I(score = 1) \mid I X_a(score = 1) \mid \varepsilon(score = 0),$$

$$X \rightarrow XI(score = 0) \mid IX(score = 0)$$

Deletion. To allow for deletion of symbols, we add for every production of the form $X \rightarrow a \in \mathcal{P}$, $X \rightarrow \varepsilon$ with a score of 1. We also add $S \rightarrow \varepsilon$ with a score of 0 to allow $\varepsilon \in \mathcal{L}(G)$.

Theorem 6 ([2], [48]). *Given a string $\sigma \in \Sigma^*$ and $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$, $d_G(G, \sigma)$ equals d if and only if σ can be parsed using G' with a minimum total score of d, where the score of a parsing is defined as the sum of the scores of the productions used in the parsing.*

Before describing the full dynamic programming, we need one more step that allows for a sequence of deletions. Define $null(A)$ to be the minimum cost of deriving $A \overset{*}{\Rightarrow} \varepsilon$ in G' if such a derivation is possible and ∞ otherwise. That is, whenever $A \overset{*}{\Rightarrow} \varepsilon$, $null(A)$ gives the length of the shortest string in Σ^* derivable from A. The next theorem from [29] (see **(B)**) shows computing $null(A)$ is easy.

Theorem 7 ([29]). *For any CFG $G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$, the set of values $\{null(A) \mid A \in \mathcal{N}\}$ can be computed in $O(|\mathcal{N}||\mathcal{P}| \log(|\mathcal{N}|))$ time.*

Let $Null$ be a list of $\{(X, null(X))$ s.t. $X \in \mathcal{N}$ and $null(X) \leq n\}$, that is it contains all non-terminals that can derive ε within at most n edits and the corresponding scores. The size of $Null$ is at most $|\mathcal{N}|$. We now create a list $Delete(A)$ for every non-terminal A iteratively. On the first iteration, we add $(A, 0)$ to $Delete(A)$. On the ith iteration, for every entry $(X, a) \in Delete(A)$, and every entry $(Y, b) \in Null$, if there exists $Z \rightarrow XY$ or $Z \rightarrow YX$ in $\mathcal{P}'$ such that $a + b + s(Z \rightarrow XY) \leq n$, then we add $(Z, a + b + s(Z \rightarrow XY))$ to $Delete(A)$ if it already does not contain an entry with non-terminal Z or if the entry with Z has a higher score (in that case we replace it). We stop when no new entry is added in an iteration. Since, the score lies in $[0, n]$ and every non-terminal can appear in $Delete(A)$ at most once with each distinct score, and $|Null| \leq |G'|$, we need $O(n\text{poly}(|G'|))$ time to construct $Delete(A)$. Hence for all non-terminals, this list can be constructed in time $O(n\text{poly}(|G'|))$. The size of $Delete(A)$ at the end is bounded by $|G'|$ for every $A \in \mathcal{N}$.

Lemma 1. *(X, c) is in $Delete(A)$ if and only if $X \overset{*}{\Rightarrow} A$ with a minimum score of c, where $A \Rightarrow A$ with a score of 0.*

Exact Dynamic Programming: The dynamic programming from [2] creates an $n \times n$ table $\mathcal{T}$ where $|\sigma| = n$. Each entry in $\mathcal{T}$ will be a list of (X, c) pairs where $X \in \mathcal{N}$ and $c \in \mathbb{Z}^+$ such that $(X, c) \in \mathcal{T}(i, j)$, $1 \le i \le j \le n$, if and only if the substring $\sigma(i : j) = \sigma_i \sigma_{i+1} ... \sigma_j$ can be derived from X with a minimum total score of c in G'. To get LED, we check $\mathcal{T}(1, n)$ and report the score corresponding to S.

Initialization. For all $i = 1, 2, .., n$, include in $\mathcal{T}(i, i)$ all non-terminals X such that $X \to a \in \mathcal{P}'$ with a score $s(X \to a)$ (assume $\sigma_i = \sigma(i : i) = a$), along with all (Y, b) such that $(Y, b - s(X \to a)) \in Delete(X)$. If any non-terminal appears multiple times, we only maintain its occurrence once with the minimum score. Note that computing $\mathcal{T}(i, i)$ requires time $O(|G'|)$. For all $\mathcal{T}(i, j), i \ne j$ initialize it with an empty list. Note that, at this point, $\mathcal{T}(i, i)$ contains (X, c) if and only if $X \overset{*}{\Rightarrow} \sigma_i$ with a minimum score of c.

Recurrence. We assume we have computed $\mathcal{T}(i, j)$ for all $i, j, i \le j$ such that $(j - i + 1) = t$, that is for all t length substrings. And, $\mathcal{T}(i, j)$ contain (X, c) if and only if $X \overset{*}{\Rightarrow} \sigma(i : j)$ with a score of c.

Now to compute $\mathcal{T}(i, j + 1)$, consider for every ℓ, $\ell = i, i+1, .., j$, $\mathcal{T}(i, \ell)$ and $\mathcal{T}(\ell+1, j+1)$. If $(X, a) \in \mathcal{T}(i, \ell)$ and $(Y, b) \in \mathcal{T}(\ell + 1, j + 1)$, and there exists a production of the form $Z \to XY$ in G' with a score of $s(Z \to XY)$, create $(Z, a + b + s(Z \to XY))$ and include $(Z, a + b + s(Z \to XY))$ to $\mathcal{T}(i, j + 1)$ if it does not already contain an entry with non-terminal Z, or if the entry with non-terminal Z in $\mathcal{T}(i, j + 1)$ has a score $> a + b + s(Z \to XY)$; in the later case replace it with the newly created entry. Next, for every (X, a) now in $\mathcal{T}(i, j + 1)$ include all (Y, b) (if already there does not exist any entry in $\mathcal{T}(i, j + 1)$ with non-terminal Y and score lower than b, and $b \le n$.) such that $(Y, b - a) \in Delete(X)$. Now, $\mathcal{T}(i, j + 1)$ contains (X, c) if and only if $X \overset{*}{\Rightarrow} \sigma(i : j + 1)$ with a minimum score of c.

Theorem 8 ([2]). *Given G and $\sigma \in \Sigma^*$, $|\sigma| = n$, if $(S, c) \in \mathcal{T}(1, n)$ then $d_G(G, \sigma) = c$ and computing it requires time $O(n^3)$ and space $O(n^2)$.*

III. Fast Additive Approximation for LED & RNA Folding

In this section, we prove Theorem 1 that provides an $O(n^2 k \log n)$ time algorithm for LED with $O(nk \log n)$ space and additive error $O(\frac{n}{k} \log n)$. This is the first combinatorial result for LED after 1972 that gets nontrivial approximation in truly subcubic time. As a corollary (Corollary 1) of Theorem 1, we can check approximate membership in CFG in quadratic time and linear-space. Moreover, we get the first combinatorial approximation for RNA folding (Corollary 2) which improves over the known bounds even using fast matrix multiplication [48] significantly.

Recall from Section II that $\mathcal{T}(i, j)$ denotes the solution computed by the exact dynamic programming for LED on substring $\sigma(i : j)$. While computing $\mathcal{T}(i, j)$, the exact algorithm considers every $\ell = i, i+1, .., j-1$ and combines the results from the already computed $\mathcal{T}(i, l)$ and $\mathcal{T}(l + 1, j)$ to obtain $\mathcal{T}(i, j)$. We refer to these $\ell \in [i, j - 1]$ as *break-points*. In the following, $\mathcal{T}_1(i, j)$ will denote the approximated answer for LED on substring $\sigma(i : j)$. While computing, $\mathcal{T}_1(i, j)$, instead of considering all break-points $\ell \in [i, j - 1]$, we will select deterministically a small set of break-points in $Break(i, j) \subseteq \{i, i + 1, .., j - 1\}$ and only consider $\ell \in Break(i, j)$. Everything else will remain the same. The choice of $Break(i, j)$ depends on a parameter $k \ge 1$, which in turn dictates the running time and the approximation factor. Let $\lceil x \rceil_{jk}$ indicates the nearest index $\ge x$ that is a multiple of jk. Similarly, $\lfloor x \rfloor_{jk}$ indicates the nearest index $\le x$ that is a multiple of jk. We call an interval $[start, end]$ valid if $start \le end$. Our approximate DP algorithm is given in Figure 1.

Observation 1. *For all $i, j \le n$, each interval L_t and $R_{t'}$ constructed by* **Approximate**-*DP^1 for $t \in [0, \tau_L(i, j)]$ and $t' \in [0, \tau_R(i, j)]$ are valid.*

Proof: Clearly L_0 is valid. Consider $L_t = [\lceil i + 2^{t-1}k \rceil_{2^{t-1} \cdot k}, \lceil i + 2^t k \rceil_{2^t \cdot k} - 1]$ for any $t \in [0, \tau_L(i, j)]$. We have $\lceil i + 2^{t-1}k \rceil_{2^{t-1} \cdot k} \le i + 2^{t-1}k + 2^{t-1}k - 1 = i + 2^t k - 1 \le \lceil i + 2^t k \rceil_{2^t \cdot k} - 1$. Hence, L_t is valid.

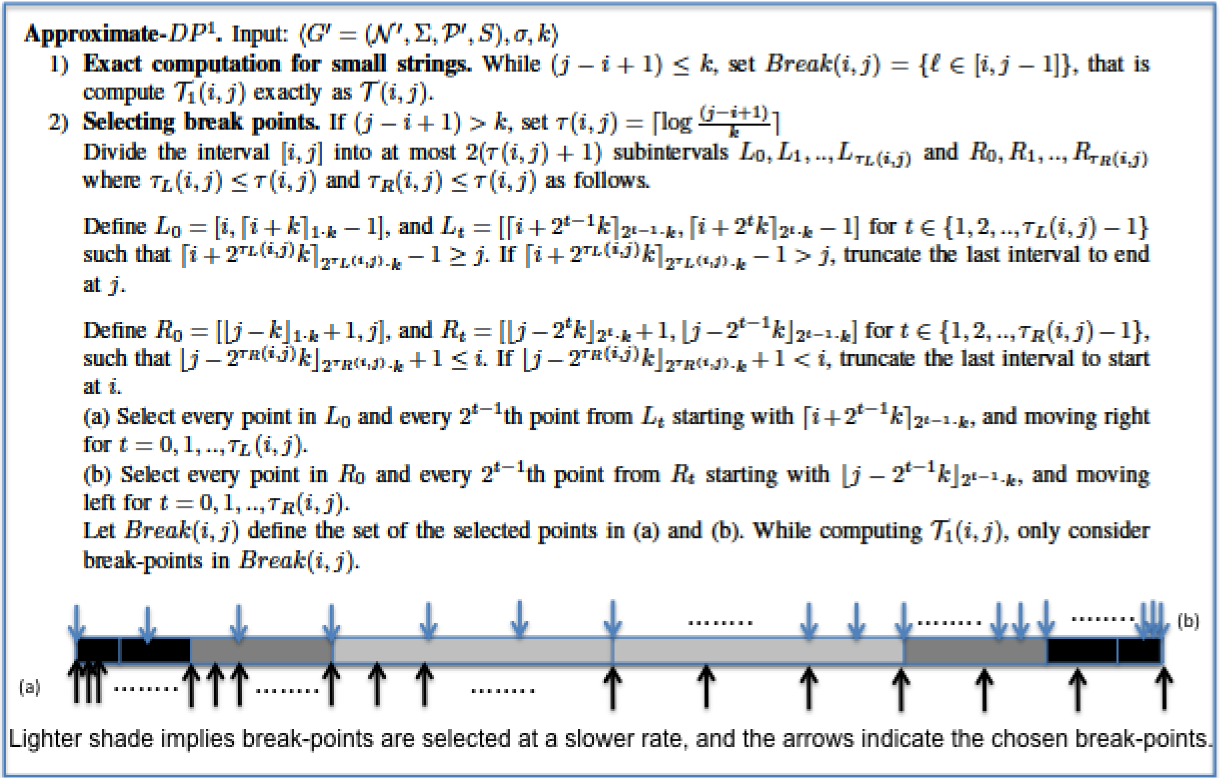
Similarly, R_0 is valid. We have $\lfloor j - 2^{t-1}k \rfloor_{2^{t-1} \cdot k} \ge j - 2^{t-1}k - (2^{t-1}k - 1) = j - 2^t k + 1 \ge \lfloor j - 2^t k \rfloor_{2^t \cdot k} + 1$. Hence R_t is valid. ■

Observation 2. *For every $\sigma(i : j)$, $\mathcal{T}_1(i : j)$ contains $(I, j - i + 1)$.*

Proof: If $j = i$, that is for length 1 substrings, we have $(X_{\sigma(i)}, 0) \in \mathcal{T}_1(i, i)$ and $(I, 0) \in Null$. Then since we have productions $I \to I X_{\sigma(i)}$ (or $I \to X_{\sigma(i)} I$), $(I, 1) \in Delete(X_{\sigma(i)})$. Thus, $(I, 1) \in T^1(i, i)$. Suppose the claim is true for all substrings of length $r - 1$, and now consider a substring of length r, $\sigma(i, i + r - 1)$. We know $i + 1 \in Break(i, i + r - 1)$ as $k \ge 1$, and $\mathcal{T}_1(i + 1, i + r - 1)$ contains $(I, r - 1)$. Now using the production $I \to X_{\sigma(i)} I$, we see $(I, r) \in \mathcal{T}_1(i, i + r - 1)$. ■

A. Proof of Theorem 1.

Lemma 2. **Approximate-**DP^1 *has a running time of $O(n^2 k \log \frac{n}{k})$.*



Approximate-DP^1. Input: $\langle G' = (\mathcal{N}', \Sigma, \mathcal{P}', S), \sigma, k\rangle$

1) **Exact computation for small strings.** While $(j - i + 1) \le k$, set $Break(i,j) = \{\ell \in [i, j-1]\}$, that is compute $\mathcal{T}_1(i,j)$ exactly as $\mathcal{T}(i,j)$.
2) **Selecting break points.** If $(j - i + 1) > k$, set $\tau(i,j) = \lceil \log \frac{(j-i+1)}{k} \rceil$
Divide the interval $[i,j]$ into at most $2(\tau(i,j)+1)$ subintervals $L_0, L_1, .., L_{\tau_L(i,j)}$ and $R_0, R_1, .., R_{\tau_R(i,j)}$ where $\tau_L(i,j) \le \tau(i,j)$ and $\tau_R(i,j) \le \tau(i,j)$ as follows.

   Define $L_0 = [i, \lceil i+k \rceil_{1.k} - 1]$, and $L_t = [\lceil i + 2^{t-1}k \rceil_{2^{t-1}.k}, \lceil i + 2^t k \rceil_{2^t.k} - 1]$ for $t \in \{1, 2, .., \tau_L(i,j) - 1\}$ such that $\lceil i + 2^{\tau_L(i,j)} k \rceil_{2^{\tau_L(i,j)}.k} - 1 \ge j$. If $\lceil i + 2^{\tau_L(i,j)} k \rceil_{2^{\tau_L(i,j)}.k} - 1 > j$, truncate the last interval to end at j.

   Define $R_0 = [\lfloor j - k \rfloor_{1.k} + 1, j]$, and $R_t = [\lfloor j - 2^t k \rfloor_{2^t.k} + 1, \lfloor j - 2^{t-1} k \rfloor_{2^{t-1}.k}]$ for $t \in \{1, 2, .., \tau_R(i,j) - 1\}$, such that $\lfloor j - 2^{\tau_R(i,j)} k \rfloor_{2^{\tau_R(i,j)}.k} + 1 \le i$. If $\lfloor j - 2^{\tau_R(i,j)} k \rfloor_{2^{\tau_R(i,j)}.k} + 1 < i$, truncate the last interval to start at i.
   (a) Select every point in L_0 and every 2^{t-1}th point from L_t starting with $\lceil i + 2^{t-1}k \rceil_{2^{t-1}.k}$, and moving right for $t = 0, 1, .., \tau_L(i,j)$.
   (b) Select every point in R_0 and every 2^{t-1}th point from R_t starting with $\lfloor j - 2^{t-1}k \rfloor_{2^{t-1}.k}$, and moving left for $t = 0, 1, .., \tau_R(i,j)$.
   Let $Break(i,j)$ define the set of the selected points in (a) and (b). While computing $\mathcal{T}_1(i,j)$, only consider break-points in $Break(i,j)$.

Figure 1: Algorithm: Approximate-DP[1]

Proof: There are $O(nk)$ substrings of length at most k; for each of which **Approximate-**DP^1 requires $O(k)$ time to compute its solution. Hence, running time for step 1 is $O(nk^2)$. For a substring $\sigma(i : j)$ of length $(j - i + 1) \in (k * 2^\eta, k * 2^{\eta+1}]$, $\eta = O(\log(n/k))$, we have $\tau(i,j) = \eta + 1$. The length of L_t, $t = 0, 1, ..., \tau_L(i,j)$ (similarly for R_t with $t = 0, 1, ..., \tau_R(i,j)$) is at most $2^{t-1}k + 2^t k < 2^{t+1}k$. While computing $\mathcal{T}_1(i,j)$, the number of break-points sampled from each L_t and similarly from each R_t is at most $4k$. Thus $|Break(i,j)| \le 8(\eta+1)k$. Hence, computing each $\mathcal{T}_1(i,j)$ with $(j - i + 1) > k$ requires time $O(k \log \frac{n}{k})$. Therefore, the overall time complexity is dominated by this step and is $O(n^2 k \log \frac{n}{k})$. ∎

We would also need the following result which will be useful for our analysis.

Lemma 3. *For any $\sigma(i,j)$ and $\sigma(i',j')$ such that $i' \ge i$ and $j' \le j$, if p is a break-point considered by $\mathcal{T}_1(i,j)$ and $p \in [i', j'-1]$, then p is also a break-point considered by $\mathcal{T}_1(i',j')$.*

Proof: Note that for any $\sigma(i,j)$ the interval $L_t(i,j)$ has a starting point that is a multiple of 2^{t-1}, except for $t = 0$. In $L_t(i,j)$, $\mathcal{T}_1$ selects every 2^{t-1}th break-point starting from the interval start-point. Thus, while in $L_t(i,j)$, all break-points that are multiples of 2^{t-1} are selected. Now consider $\sigma(i',j')$ which is a substring of $\sigma(i,j)$. Let p be a break-point selected by $\sigma(i,j)$. Suppose $p \in [i', j'-1]$ and also $p \in L_t(i,j)$ for some t when it is selected. Then $p \in L_{t'}(i',j')$ for some $t' \le t$. If $t = 0$, then $t' = 0$ as well and p is in $Break(i',j')$. Else, since p is a multiple of 2^{t-1}, it is also a multiple of $2^{t'-1}$, therefore again $p \in Break(i',j')$.

If $p \in R_t(i,j)$ for some t when it is selected, again following the construction of R_t and the selected points, p is a multiple of 2^{t-1}, and $p \in R_{t'}(i',j')$ for some $t' \le t$, and it gets selected in $R_{t'}(i',j')$. ∎

Corollary 3. *If p is a break-point considered by $\mathcal{T}_1(i,j)$ such that $p \in [i+1, j-2]$, then p is also a break-point considered by both $\mathcal{T}_1(i+1,j)$ and $\mathcal{T}_1(i,j-1)$.*

$\mathcal{T}(i,j)$ and similarly $\mathcal{T}_1(i,j)$ implicitly maintain the entire parsing information for substring $\sigma(i,j)$ to obtain the minimum cost parse tree starting with every non-terminal. Let us use $P^*_Z(i,j)$ to designate the parse tree obtained from $\mathcal{T}(i,j)$ for $Z \overset{*}{\Rightarrow} \sigma(i : j)$ and $cost(P^*_Z(i,j))$ be the total cost of the deriving it. Similarly, $P^1_Z(i,j)$ and $cost(P^1_Z(i,j))$ denote the parse

tree and its cost obtained from $\mathcal{T}_1(i,j)$ for $Z \stackrel{*}{\Rightarrow} \sigma(i:j)$. To show how these parse trees are constructed, we need to consider the following cases.

Case 0. We include (Z,c) in $\mathcal{T}(i,j)$ where Z produces a terminal, then Z is the root node of the parse tree which is also a leaf node.

Case 1. We include (Z,c) in $\mathcal{T}(i,j)$ by combining $(X,a) \in \mathcal{T}(i,l)$ and $(Y,b) \in \mathcal{T}(l+1,j)$, $l \in [i,j-1]$, then Z is the root node of the parse tree. We proceed creating its left child starting with $(X,a) \in \mathcal{T}(i,l)$ and right child starting with $(Y,b) \in \mathcal{T}(l+1,j)$.

Case 2. We include (Z,c) in $\mathcal{T}(i,j)$ by combining $(X,a) \in \mathcal{T}(i,j)$ and some $(Y, null(Y))$ that is $(Z, c-a-null(Y)+s(Z \rightarrow XY)) \in Delete(X)$, then we create a vertex Z, associate X and Y as its two children and proceed with expanding $(X,a) \in \mathcal{T}(i,j)$. We continue until we reach at a vertex (Z',c') in $\mathcal{T}(i,j)$ by combining $(X',a') \in \mathcal{T}(i,l)$ and $(Y',b') \in \mathcal{T}(l+1,j)$, $l \in [i,j-1]$. The entire subtree starting with Z (root of that subtree) and ending at Z' becomes the designated root node of the parse tree. To differentiate between the "node" of this subtree from node of the parse tree, we refer to them as vertices. We proceed creating its left child starting with $(X',a') \in \mathcal{T}(i,l)$ and right child starting with $(Y',b') \in \mathcal{T}(l+1,j)$.

At every node of a parse tree, we maintain the non-terminal corresponding to the root of the subtree represented by that node, the production used to create its two children and the cost for using that production plus the cost to generate the subtree represented by that node. If we expand the vertices that are created by considering $Delete()$ in Case 2, then every node v in a parse tree has two children designated by v_L (left child) and v_R (right child) unless it produces a terminal (in which case it is a leaf node). Naturally, each node of a parse tree generates a substring given by the leaf nodes beneath it. $w(v)$ denotes the length of the substring generated by v with $w(v)=0$ if it generates ε. The total cost of a parse tree is the sum of costs of all its nodes.

In a similar way, we can generate the parse tree $P^1_Z(i,j)$. It is useful to think while $\mathcal{T}(i,j)$ considers all possible parse trees for $\sigma(i,j)$ and selects the one with minimum cost for every non-terminal, $\mathcal{T}_1(i,j)$ considers a subset of the parse trees restricted by the choice of the break-points, and selects the one with minimum cost from that subset, again for every non-terminal. Consider Case 2 where (Z,c) and (Z',c') are in $\mathcal{T}(i,j)$. If $(Z',c'_1) \in \mathcal{T}_1(i,j)$, then we must also have $(Z, c'_1+[c-c']) \in \mathcal{T}_1(i,j)$, and the exact same subtree starting with Z and ending at Z' becomes the designated root node for $P^1_Z(i,j)$. Therefore, to relate the cost of $cost(P^1_Z(i,j))$ and $cost(P^*_Z(i,j))$, it is enough to relate the cost of $cost(P^1_{Z'}(i,j))$ and $cost(P^*_{Z'}(i,j))$ in this case and show whenever, $\mathcal{T}(i,j)$ maintains a parse tree starting with a non-terminal $Z' \in \mathcal{N}$, that is $P^*_{Z'}(i,j)$, $\mathcal{T}_1(i,j)$ also maintains a parse tree $P^1_{Z'}(i,j)$ with a cost not significantly higher than $cost(P^*_{Z'}(i,j))$. For simplicity of notation, we often drop the subscript and use $P^*(i,j)$ and $P^1(i,j)$ to denote the parse trees respectively for $\mathcal{T}(i,j)$ and $\mathcal{T}_1(i,j)$ starting with the same non-terminal.

Lemma 4. *For any $\sigma(i,j)$ and $\sigma(i',j')$ such that $i' \geq i$ and $j' \leq j$, $cost(P^1(i',j')) \leq cost(P^1(i,j)) + |i'-i| + |j'-j|$.*

Proof: We exhibit a parse tree $P'(i',j')$ for $\sigma(i',j')$ which is within the subset of parse trees considered by $\mathcal{T}_1$ and closely mimics $P^1(i,j)$, and has the desired cost. If the root of $P^1(i,j)$ is v, create the root of $P'(i',j')$ as $v'=v$. Let v_L produce $\sigma(i,\ell)$ and v_R produce $\sigma(\ell+1,j)$. If $\ell \in [i',j'-1]$, then set v'_L as v_L and v'_R as v_R. Proceed with both v_L and v_R. Otherwise, $\ell \notin [i',j'-1]$, then one of v_L or v_R produces a substring of $\sigma(i,i'-1)$ or $\sigma(j'+1,j)$. Say v_L produces a substring in $\sigma(i,i'-1)$ (the case for v_R producing a substring of $\sigma(j'+1,j)$ is similar) with a cost of c_L, then v_L can produce ε with a cost of $c_L+\eta$, where η is the length of the substring of $\sigma(i,i'-1)$ that it produces. We map v'_R to v_R and v'_L to $\varepsilon(v_L)$ where $\varepsilon(v_L)$ corresponds to a node that starting with the same nonterminal as v_L produces ε with a cost of $null(v_L)$. Note that $null(v_L) \leq c_L+\eta$. We then just proceed with v_R. At the end we have a parse tree $P'(i',j')$ that generates $\sigma(i',j')$ such that for every non-leaf node with none of its two children deriving ε, it contains a break-point in $[i',j'-1]$ considered by $P^1(i,j)$. Thus these break-points are also considered by $\mathcal{T}_1$ while computing $P^1(i',j')$ by Lemma 3. Hence $cost(P^1(i',j')) \leq cost(P'(i',j'))$. But $cost(P'(i',j')) \leq cost(P^1(i,j)) + |i'-i| + |j'-j|$. Hence, $cost(P^1(i',j')) \leq cost(P^1(i,j)) + |i'-i| + |j'-j|$. ■

Let us refer to a vertex $v \in P^*(i,j)$ (similarly in $P^1(i,j)$) as internal, if it has two children and none of them produces ε. Every non-leaf node in the parse tree contains exactly one internal vertex (the vertex Z in Case 1, and the leaf vertex Z' in Case 2). Let $Min(i,j) = \sum_{v \in V^*(i,j)} \min \lfloor \frac{w(v_L)}{k} \rfloor, \lfloor \frac{w(v_R)}{k} \rfloor$ where $V^*(i,j)$ are all the non-leaf nodes in the parse tree $P^*(i,j)$.

Lemma 5. *For all $i,j, i \leq j \in [1,n]$, there exists*

a parse tree $P^1(i,j)$ maintained by $\mathcal{T}_1(i,j)$ such that $cost(P^1(i,j)) \leq cost(P^*(i,j)) + 4 \cdot Min(i,j)$.

Proof: The proof is by induction on $w(v)$, where v generates the substring $\sigma(i:j)$, that is $w(v) = (j - i + 1)$.

Base case. When $w(v) \leq k$, since $\mathcal{T}_1(i,j)$ is computed by exactly following $\mathcal{T}(i,j)$, $P^1(v)$ is same as $P^*(v)$. Hence $cost(P^1(v)) = cost(P^*(v))$. Thus the base case satisfies the claim.

Induction hypothesis. We assume that by the induction hypothesis, the claim is true for all $\sigma(i,j)$ such that $j - i + 1 \leq kt$, $t \geq 1$.

Induction. We now consider a substring $\sigma(i,j)$ such that $j - i + 1 \in [kt+1, k(t+1)]$. Suppose $\mathcal{T}(i,j)$ considers the break-point $\ell \in [i, j-1]$ to generate $P^*(i,j)$ using the production $Z \to XY$, where X generates $\sigma(i,\ell)$ and Y generates $\sigma(\ell+1,j)$. If either of X or Y generates a substring of length at most k, then $\ell \in Break(i,j)$, and we recursively consider the child generating the larger substring until we reach at a node where both its children generate substrings larger than k.

We overuse symbols to keep the notation simple, and assume without loss of generality that both X and Y generate substrings of length strictly larger than k. Also, without loss of generality, let us assume that $w(v_L) \leq w(v_R) \leq kt$; the other case will be symmetric.

We have $w(v_L) = (\ell - i + 1)$. Let $k2^{a-1} + 1 \leq w(v_L) \leq k2^a$ for some $a \geq 1$.

Let the nearest break-point to the left of ℓ in $Break(i,j)$ while parsing $\sigma(i:j)$ be ℓ'. Then it must hold that $|\ell - \ell'| \leq 2^{a-1} = \lfloor \frac{w(v_L)}{k} \rfloor$. So $\mathcal{T}_1(i,j)$ checks the possibility of combining $\mathcal{T}_1(i:\ell')$ and $\mathcal{T}_1(\ell'+1,j)$.

By induction hypothesis on v_L and v_R, we have

$$cost(P^1[i,\ell]) \leq cost(P^*[i,\ell]) + 4 \cdot Min(i,\ell) \text{ and}$$

$$cost(P^1[\ell+1,j]) \leq cost(P^*[\ell+1,j]) + 4 \cdot Min(\ell+1,j)$$

We would like to use parse trees that closely mimic $P^1[i,\ell]$ to generate $\sigma(i,\ell')$ and $P^1(l+1,j)$ to generate $\sigma(\ell'+1,j)$ by paying minimum amount of additional cost. That will give us the desired approximation.

Generating parse trees for $\sigma(i,\ell')$ and $\sigma(\ell'+1,j)$. Suppose to generate $P^1[i,\ell]$, $\mathcal{T}(i,\ell)$ considers break-point $\ell_1 \in [i, \ell-1]$ to derive its left and right child using non-terminals X^1 and Y^1 respectively.

Claim 1. $cost(P^1[i,\ell']) \leq 2^{a-1} + cost(P^1[i,\ell])$.

Proof. **Case 1.** $\ell_1 \in [i,\ell')$. Then $\mathcal{T}_1(i,\ell')$ can use $P^1_{X^1}[i,\ell_1]$ to derive $\sigma(i,\ell_1)$ and use $P^1_{Y^1}[\ell_1+1,\ell']$ to derive $\sigma(i,l')$ with a cost at most $cost(P^1_{Y^1}[\ell_1+1,\ell]) + |\ell - \ell'|$ from Lemma 4. Thus, $cost(P^1[i,\ell']) \leq |\ell - \ell'| + cost(P^1[i,\ell]) \leq 2^{a-1} + cost(P^1[i,\ell])$.

Case 2. $\ell_1 \geq \ell'$. Then ℓ_1 is not a valid break-point to be considered for generating $\sigma(i:\ell')$. But $P^1_{Y^1}[\ell_1+1,\ell]$ can generate the empty string with a cost of $cost(P^1_{Y^1}[\ell_1+1,\ell]) + |\ell_1 - \ell|$. Then $(Y_1, c_1) \in Null$ where $c_1 \leq cost(P^1_{Y^1}[\ell_1+1,\ell]) + |\ell_1 - \ell|$. Moving to $P^1_{X_1}[i,\ell_1]$, if it considers the break-point ℓ_2 to derive its two children using non-terminals X^2 and Y^2 which is again $\geq \ell'$, then we can use $P^1_{Y^2}[\ell_2+1,\ell_1]$ to generate the empty string with a cost of $cost(P^1_{Y^2}[\ell_2+1,\ell_1]) + |\ell_2 - \ell_1|$. We have $(Y^2, c_2) \in Null$ where $c_2 \leq cost(P^1_{Y^2}[\ell_2+1,\ell_1]) + |\ell_2 - \ell_1|$. Otherwise, $\ell_2 < \ell'$, then we use $P^1_{X^2}[i,\ell_2]$ to generate $\sigma(i,\ell_2)$ and $P^1_{Y^2}[\ell_2+1,\ell']$ to generate $\sigma(\ell_2+1,\ell')$ with a cost of $cost(P^1_{Y^2}[\ell_2+1,\ell_1]) + |\ell' - \ell_1|$. Thus, $cost(P^1_{X_1}(i,\ell')) \leq cost(P^1_{X_1}(i,\ell_1)) + |\ell_1 - \ell'|$. Finally, this can be combined with (Y_1, c_1) via $Delete(X_1)$ to generate the parse tree for $\sigma(i,\ell')$ starting with the same non-terminal as $P^1[i,\ell]$ with a total cost of at most $|\ell - \ell'| + cost(P^1[i,\ell])$. Continuing in this manner, we again get,

$$\begin{aligned} cost(P^1[i,\ell']) &\leq |\ell - \ell'| + cost(P^1[i,\ell]) \\ &\leq 2^{a-1} + cost(P^1[i,\ell]). \quad \square \end{aligned}$$

Claim 2. $cost(P^1[\ell'+1,j]) \leq 2^a + cost(P^1[\ell+1,j])$

Proof. To generate $P^1[\ell+1,j]$, suppose $\mathcal{T}_1(\ell+1,j)$ considers break-point $\ell_R \in [\ell+1, j-1]$ to derive its two children. Then ℓ_R must be $\in [\ell'+1, j)$ always. Consider, the nearest break-point to the right of ℓ in $Break(i,j)$, denoted by ℓ''. We can argue exactly as in Claim 1 to get

$$cost(P^1_Y[\ell''+1,j]) \leq |\ell - \ell''| + cost(P^1_Y[\ell+1,j]).$$

Since $\sigma(\ell'+1,j)$ is a substring of $\sigma(i,j)$, we have from Lemma 3 that $\mathcal{T}_1(\ell'+1,j)$ considers the break-point ℓ''. But, then $\mathcal{T}_1$ considers a parse tree for $\sigma(\ell'+1,j)$ that uses the production $Y \to IY$ at the root and has $I \stackrel{*}{\Rightarrow} \sigma(\ell',\ell'')$ with a cost of $|\ell' - \ell''| \leq 2^{a-1}$ as its left child and $P^1_Y[\ell''+1,j]$ as its right. Thus, $cost(P^1[\ell'+1,j]) \leq 2^a + cost(P^1[\ell+1,j])$, since both $|\ell - \ell''|$ and $|\ell' - \ell''|$ are $\leq 2^{a-1}$. $\square$

Now, $4 \cdot [Min(i,l) + Min(l+1,j)] + 3 \cdot 2^{a-1} < 4 \cdot Min(i,j)$, thus we have the total cost to be

$$cost(P^1[i,j]) \leq cost(P^*[i,j]) + 4 \cdot Min(i,j).$$

■

We now prove a structural lemma, which shows that

$$Min[i,j] \leq \frac{(j-i+1)}{k} \log{(j-i+1)},$$

giving the desired approximation.

Suppose we are given a rooted binary tree T with n leaves. Each node v is assigned a weight $w(v)$ which corresponds to the number of leaves in the subtree rooted at v. Let v_L denote the left child of v and v_R denote the right child of v. We are interested in having a bound on $\sum_{v:\text{non-leaf}\in T} \min(w(v_L), w(v_R))$. For any node v which has only one child or is a leaf, define $\min(w(v_L), w(v_R)) = 0$

Lemma 6. *For any rooted binary tree T with n leaves and weight w on nodes defined as above,*

$$\sum_{v\in T} \min(w(v_L), w(v_R)) \leq n \log n.$$

Combining Lemma 5 and Lemma 6, we get the desired approximation factor of $\frac{n}{k}\log n$. Lemma 2 guarantees a running time of $O(n^2 k \log n)$. However, the space required is $O(n^2)$. In the full version, we show the space usage can be further reduced to $O(nk \log n)$, thus completing the proof of Theorem 1.

IV. Longest Increasing Subsequence

In this section, we give a short proof which gives a deterministic additive ϵn-approximation for the longest increasing subsequence (LIS) in $O(\frac{\log n}{\epsilon})$ space, improving the result of [50] by a $\log n$ factor. Moreover, the same proof can be easily adapted to give a multiplicative $(1+\epsilon)$-approximation to distance to monotonicity.

Given a sequence $s[1], s[2], ..., s[n]$, our goal is to find a sequence $1 \leq i_1 \leq i_2 \leq ....i_l \leq n$ such that $s[i_1] \leq s[i_2] \leq .... \leq s[i_l]$ and l is maximized. The following simple dynamic programming finds the length of the longest increasing sequence of $s[1], s[2], .., s[i]$ of $LIS[i]$.

- Initialization: $LIS[1] = 1$
- Recursion:

$$LIS[l] = \max_{r\leq l} LIS[r] + \mathcal{I}(s[r] \leq s[l])$$

where $\mathcal{I}(s[r] \leq s[l])$ is 1 if $s[r] \leq s[l]$ and 0 otherwise. Thus, each $r \leq l$ is considered as a break-point.

We choose a parameter k, and follow **Approximate-**DP^1, for every $i \in [1, n]$, and construct the intervals $R_0, R_2, ..., R_{\tau(i,0)^R}$ for $(0, i)$. That is $R_0 = [\lfloor i-k \rfloor_1 + 1, i]$, and $R_t = [\lfloor i-2^t k\rfloor_{2^t} + 1, \lfloor i-2^{t-1}k\rfloor_{2^{t-1}}]$ for $t \in \{1, 2, .., \tau(0,i)^R - 1\}$, such that $\lfloor i-2^{\tau(0,i)^R}k\rfloor_{2^{\tau(0,i)^R}} + 1 \leq 0$. If $\lfloor i - 2^{\tau(0,i)^R}k\rfloor_{2^{\tau(0,i)^R}} + 1 < 0$, truncate the last interval to start at 0.

Sample every point in R_0 and every 2^{t-1}th point from R_t starting with $\lfloor i - 2^{t-1}k\rfloor_{2^{t-1}}$, and progressively sampling lower indexed values from R_t for $t = 0, 1, .., \tau(i,j)^R$. Let $Break(i)$ define these sampled points.

Change the exact dynamic programming as follows.

- Initialization: $LIS^1[1] = 1$
- Recursion: $LIS^1[i] = \max_{r\in Break[i]} LIS^1[r] + \mathcal{I}(s[r] \leq s[i])$

From Lemma 3, the break-points considered by $LIS^1[i]$ is a subset of break-points considered by $LIS^1[i-1]$ and we need the answer computed for $LIS^1[i-1]$ to compute $LIS^1[i]$. Thus the total space usage is $O(k \log \frac{n}{k})$.

Now for the approximation factor, let $P(i)$ define the actual sequence computed by the exact dynamic programming via backtracking, and $cost(P(i))$ define the number of symbols in $s[1]s[2]....s[i]$ that is not in the LIS (formally known as the distance to monotonicity). Similarly, let $P^1(i)$ define the actual sequence computed by the approximate dynamic programming via backtracking, and $cost(P^1(i))$ define the number of symbols in $s[1]s[2]....s[i]$ that is not in this approximate LIS.

Lemma 7. *For any $i \in [1, n]$, $cost(P^1(i)) \leq cost(P(i)) + \lfloor \frac{i}{k} \rfloor$.*

Proof: The proof is by induction on i.

Base case. $i \leq k$, in that case $LIS^1[i] = LIS[i]$. Hence the base case is satisfied. By induction hypothesis assume the claim is true for all $i \leq kt$.

Induction. Now consider $i \in [kt+1, k(t+1)]$. Suppose $LIS[i]$ considers a break-point $\ell \in [1, i-1]$. If $(i - \ell + 1) \leq k$, then $\ell \in Break(i)$ and we move to $LIS[\ell]$. If $LIS[\ell]$ considers break-point ℓ^1, such that $(\ell - \ell_1 + 1) \leq k$, then again $\ell_1 \in Break(\ell)$, and we move to $LIS[\ell^1]$. We continue until we reach at a point x such that $LIS[x]$ considers a breakpoint which is more than k away from x.

Let the break-point considered by $LIS[x]$ is ℓ_x and $k2^{a-1} + 1 \leq (x - \ell_x) \leq k2^a$.

Let the nearest break-point to the left of ℓ_x is f, then it must hold that $(\ell_x - f) \leq 2^{a-1}$.

Now, by induction hypothesis we know, $cost(P^1[\ell_x]) \leq cost(P[\ell_x]) + 2\lfloor \frac{\ell_x}{k} \rfloor$. However note that any break-point considered by $LIS^1[\ell_x]$ within $[1, f]$ is also considered by $LIS^1[f]$. In fact, any break-point considered by $s(1:j)$ for any j in $[1, j']$ is also considered by $s(1:j')$, $j' \leq j$. Therefore,

Case 1: If $s[\ell_x] \geq s[f]$ then

$$LIS^1[\ell_x] \leq LIS^1[f] + (\ell_x - f),$$

$$\text{or, } cost(P^1(f)) \leq cost(P^1(\ell_x)).$$

Case 2: If $s[\ell_x] < s[f]$ then

$$LIS^1[\ell_x] \leq LIS^1[f] + (\ell_x - f) - 1,$$

$$\text{or, } cost(P^1(f)) \leq cost(P^1(\ell_x)) - 1.$$

Now,

$$cost(P^1(x)) \\ \leq (x - f - 1) + cost(P^1(f)) + (1 - \mathcal{I}(s[f] \leq s[x]))$$

$$cost(P(x)) \\ = (x - \ell_x - 1) + cost(P(\ell_x)) + (1 - \mathcal{I}(s[\ell_x] \leq s[x]))$$

For Case 1, $\mathcal{I}(s[f] \leq s[x]) \geq \mathcal{I}(s[\ell_x] \leq s[x])$ and we have

$$\begin{aligned} & cost(P^1(x)) \\ & \leq (x - \ell_x - 1) + (\ell_x - f) + cost(P^1(\ell_x)) \\ & + (1 - \mathcal{I}(s[f] \leq s[x])) \\ & \leq (x - \ell_x - 1) + (\ell_x - f) + (1 - \mathcal{I}(s[\ell_x] \leq s[x])) \\ & + cost(P[\ell_x]) + \lfloor \frac{\ell_x}{k} \rfloor \end{aligned}$$

Hence, we have,

$$\begin{aligned} cost(P^1(x)) & \leq cost(P(x)) + (\ell_x - f) + \lfloor \frac{\ell_x}{k} \rfloor \\ & \leq cost(P(x)) + \lfloor \frac{x - \ell_x}{k} \rfloor + \lfloor \frac{\ell_x}{k} \rfloor \\ & \leq cost(P(x)) + \lfloor \frac{x}{k} \rfloor \end{aligned}$$

For Case 2,

$$\begin{aligned} & cost(P^1(x)) \\ & \leq (x - \ell_x - 1) + (\ell_x - f) + cost(P^1(\ell_x)) - 1 \\ & + (1 - \mathcal{I}(s[f] \leq s[x])) \\ & \leq (x - \ell_x - 1) + (\ell_x - f) + (1 - \mathcal{I}(s[\ell_x] \leq s[x])) \\ & + cost(P[\ell_x]) + \lfloor \frac{\ell_x}{k} \rfloor \\ & \leq cost(P(x)) + \lfloor \frac{x}{k} \rfloor \end{aligned}$$

And finally, $cost(P^1(i)) \leq cost(P(i)) + \lfloor \frac{i}{k} \rfloor \leq cost(P(i)) + \lfloor \frac{i}{k} \rfloor$, proving the lemma. ■

Theorem 9. *There exists a deterministic algorithm for LIS that uses $O(\frac{\log n}{\epsilon})$ space and computes an ϵn-additive approximation of LIS.*

Acknowledgement.: The author thanks the anonymous reviewers for many helpful comments about writing. This work is partially supported by a NSF CAREER 1652303 grant, a NSF CCF 1464310 grant, a Yahoo ACE Award and a Google Faculty Research Award.

REFERENCES

[1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is valiant's parser. In *FOCS*, pages 98–117, 2015.

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

[3] David Aldous and Persi Diaconis. Longest increasing subsequences: from patience sorting to the baik-deift-johansson theorem. *Bulletin of the American Mathematical Society*, 36(4):413–432, 1999.

[4] Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. Regular languages are testable with a constant number of queries. *SIAM J. Comput.*, 30(6), December 2001.

[5] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *FOCS*, pages 377–386, 2010.

[6] Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. In *STOC*, pages 199–204, 2009.

[7] John Aycock and R Nigel Horspool. Practical earley parsing. *The Computer Journal*, 45(6):620–630, 2002.

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

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

[10] Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *FOCS*, pages 550–559, 2004.

[11] Tuğkan Batu, Funda Ergun, and Cenk Sahinalp. Oblivious string embeddings and edit distance approximations. In *SODA*, pages 792–801, 2006.

[12] Djamal Belazzougui and Qin Zhang. Edit distance: Sketching, streaming, and document exchange. In *FOCS*, pages 51–60, 2016.

[13] Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. Truly sub-cubic algorithms for language edit distance and rna-folding via fast bounded-difference min-plus product. In *FOCS*, pages 375–384, 2016.

[14] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *STOC*, pages 712–725, 2016.

[15] Richard Cole and Ramesh Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM J. Comput.*, 31(6), June 2002.

[16] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F Italiano. Sparse dynamic programming i: linear cost functions. *Journal of the ACM (JACM)*, 39(3):519–545, 1992.

[17] Michael L Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.

[18] Daniel Fredouille and Christopher H Bryant. Speeding up parsing of biological context-free grammars. In *Annual Symposium on Combinatorial Pattern Matching*, pages 241–256. Springer, 2005.

[19] François Le Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303, 2014.

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

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

[22] R.R Gutell, J.J. Cannone, Z Shang, Y Du, and M.J Serra. A story: unpaired adenosine bases in ribosomal RNAs. In *Journal of Mol. Biology*, volume 304, pages 335–354, 2010.

[23] Yijie Han and Tadao Takaoka. An $o(n^3loglogn/log^2n)$ time algorithm for all pairs shortest paths. In *SWAT*. 2012.

[24] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation, 2Nd Edition*. ACM, 2001.

[25] Rajesh Jayaram and Barna Saha. Approximating language edit distance beyond fast matrix multiplication: Ultralinear grammars are where parsing becomes hard! In *ICALP*, pages 19:1–19:15, 2017.

[26] Mark Johnson. PCFGs, Topic Models, Adaptor Grammars and Learning Topical Collocations and the Structure of Proper Names. In *ACL*, pages 1148–1157, 2010.

[27] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.

[28] Donald E Knuth. Optimum binary search trees. *Acta Informatica*, 1(3):270–270, 1972.

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

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

[31] Andreas Krebs, Nutan Limaye, and Srikanth Srinivasan. Streaming algorithms for recognizing nearly well-parenthesized expressions. In *MFCS*, 2011.

[32] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM J. Comput.*, 27(2), April 1998.

[33] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA*, pages 85–94. ACM, 2011.

[34] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1), January 2002.

[35] Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. In *STOC*, 2010.

[36] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18 – 31, 1980.

[37] Darnell Moore and Irfan Essa. Recognizing multitasked activities from video using stochastic context-free grammar. In *NCAI*, pages 770–776, 2002.

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

[39] Timothy Naumovitz and Michael Saks. A polylogarithmic space deterministic streaming algorithm for approximating distance to monotonicity. In *SODA*, pages 1252–1262, 2015.

[40] Ruth Nussinov and Ann B. Jacobson. Fast algorithm for predicting the secondary structure of single-stranded rna. *Proceedings of the National Academy of Sciences of the United States of America*, 77(11):6309–6313, 1980.

[41] Rafail Ostrovsky and Yuval Rabani. Low distortion embeddings for edit distance. *J. ACM*, 54(5), October 2007.

[42] Michal Parnas, Dana Ron, and Ronitt Rubinfeld. Testing membership in parenthesis languages. *Random Struct. Algorithms*, 22(1), January 2003.

[43] Terence Parr and Kathleen Fisher. Ll (*): the foundation of the antlr parser generator. *ACM SIGPLAN Notices*, 46(6):425–436, 2011.

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

[45] Stefano Crespi Reghizzi, Luca Breveglieri, and Angelo Morzenti. *Formal languages and compilation*, volume 795. Springer, 2009.

[46] Andrea Rosani, Nicola Conci, and Francesco G. De Natale. Human behavior recognition using a context-free grammar. *Journal of Electronic Imaging*, 23(3), 2014.

[47] Barna Saha. The dyck language edit distance problem in near-linear time. In *FOCS*, pages 611–620, 2014.

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

[49] Michael Saks and C Seshadhri. Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance. In *SODA*, pages 1698–1709, 2013.

[50] Michael E. Saks and C. Seshadhri. Estimating the longest increasing sequence in polylogarithmic time. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 458–467, 2010.

[51] Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975.

[52] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, pages 887–898, 2012.

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

[54] Ye-Yi Wang, Milind Mahajan, and Xuedong Huang. A unified context-free grammar and n-gram model for spoken language processing. In *ICASP*, pages 1639–1642, 2000.