# An error correcting parser for context free grammars that takes less than cubic time

Sanguthevar Rajasekaran and Marius Nicolae Department of CSE, Univ. of Connecticut, Storrs

#### Abstract

The problem of parsing has been studied extensively for various formal grammars. Given an input string and a grammar, the parsing problem is to check if the input string belongs to the language generated by the grammar. A closely related problem of great importance is one where the input are a string  $\mathcal{I}$  and a grammar G and the task is to produce a string  $\mathcal{I}'$  that belongs to the language generated by G and the 'distance' between  $\mathcal{I}$  and  $\mathcal{I}'$  is the smallest (from among all the strings in the language). Specifically, if  $\mathcal{I}$  is in the language generated by G, then the output should be  $\mathcal{I}$ . Any parser that solves this version of the problem is called an error correcting parser. In 1972 Aho and Peterson presented a cubic time error correcting parser for context free grammars. Since then this asymptotic time bound has not been improved under the (standard) assumption that the grammar size is a constant. In this paper we present an error correcting parser for context free grammars that runs in O(T(n)) time, where n is the length of the input string and T(n) is the time needed to compute the tropical product of two  $n \times n$  matrices.

In this paper we also present an  $\frac{n}{M}$ -approximation algorithm for the language edit distance problem that has a run time of  $O(Mn^{\omega})$ , where  $O(n^{\omega})$  is the time taken to multiply two  $n \times n$  matrices. To the best of our knowledge, no approximation algorithms have been proposed for error correcting parsing for general context free grammars.

# 1 Introduction

Parsing is a well studied problem owing to its numerous applications. For example, parsing finds a place in programming language translations, description of properties of semistructured data [12], protein structures prediction [11], etc. For context free grammars, two

classical algorithms can be found in the literature: CYK [3, 14, 6] and Earley [4]. Both of these algorithms take  $O(n^3)$  time in the worst case. Valiant has shown that context free recognition can be reduced to Boolean matrix multiplication [13].

The problem of parsing with error correction (also known as the language edit distance problem) has also been studied well. Also and Peterson presented an  $O(n^3)$  time algorithm for context free grammar parsing with errors. Three kinds of errors were considered, namely, insertion, deletion, and substitution. This algorithm depended quadratically on the size of the grammar and was based on Earley parser. Subsequently, Myers [9] presented an algorithm for error correcting parsing for context free grammars that also runs in cubic time but the dependence on the grammar size was linear. This algorithm is based on the CYK parser.

As far as the worst case run time is concerned, to the best of our knowledge, cubic time is the best known for the error correcting parsing problem for general context free grammars. A number of approximation algorithms have been proposed for the DYCK language (which is a very specific context free language). See e.g., [12].

In this paper we present a cubic time algorithm for error correcting parsing that is considerably simpler than the algorithms of [1] and [9]. This algorithm is based on the CYK parser. Even though the algorithm of [9] is also based on CYK parser, there are some crucial differences between our algorithm and that of [9]. We also show that the language edit distance problem can be reduced to the problem of computing the tropical product (also known as the distance product or the min-plus product) of two given  $n \times n$  matrices where  $n = |\mathcal{I}|$ . Using the current best known run time [2] for tropical matrix product, our reduction implies that the language edit distance problem can be solved exactly in  $O\left(\frac{n^3(\log\log n)^3}{(\log n)^2}\right)$  time, improving the cubic run time that has remained the best since 1972.

In many applications, it may suffice to solve the language edit distance problem approximately. To the best of our knowledge, no approximation algorithms are known for general context free grammars. However, a number of such algorithms have been proposed for the Dyck language. Dyck language is a basic and fundamental context free grammar and the Dyck language edit distance is a significant generalization of string edit distance problem which has been widely studied. Also, the non-deterministic Dyck language is the hardest context free grammar in terms of parsing. The concept of approximate error correcting parsing was introduced by [7]. The algorithm of [7] takes subcubic time but its approximation factor is  $\Theta(n)$ . If  $\mathcal{I}'$  is a string in the language generated by the input grammar G that has the minimum language edit distance (say d) with the input string  $\mathcal{I}$  and if an algorithm  $\mathcal{A}$  outputs a string  $\mathcal{I}''$  such that the language edit distance between  $\mathcal{I}$  and  $\mathcal{I}''$  is no more than  $d\beta(n)$ , then we say that  $\mathcal{A}$  is a  $\beta(n)$ -approximation algorithm. Saha [12] gives the

very first near-linear time algorithm for Dyck language edit distance problem with polylog approximation and an  $O(n^{1+\epsilon} + n^{\epsilon}d^2)$  time algorithm with  $1/\epsilon \log(d)$  approximation where d is the edit distance. [12] not only studies Dyck language edit distance problem, but also a larger class of problems including the memory checking languages, which comprise of transcripts of any popular data structure. It can also be applied to other variants of languages studied by Parnas, Ron and Rubinfeld (APPRO-RANDOM, 2001). It is noteworthy that Dyck grammar parsing (without error correction) can easily be done in linear time. On the other hand, it is known that parsing of arbitrary context free grammars is as difficult as boolean matrix multiplication [8]. For an extensive discussion on approximation algorithms for the Dyck language, please see [12]. In this paper we present an approximation algorithm for general context free grammars. Specifically, we show that if we are only interested in edit distances of no more than M, then the language edit distance problem can be solved in  $O((M+1)n^{\omega})$  time where  $O(n^{\omega})$  is the time taken to multiply two  $n \times n$  matrices. (Currently the best known value for  $\omega$  is 2.376). As a corollary, it follows that there is an  $\frac{n}{M}$ -approximation algorithm for the language edit distance problem with a run time of  $O(Mn^{\omega}).$ 

### 1.1 Some Notations

A context free grammar G is a 4-tuple (N, T, P, S), where T is a set of characters (known as terminals) in the alphabet, N is a set of variables known as nonterminals, S is the start symbol (that is a nonterminal) and P is a set of productions.

We use L(G) to denote the language generated by G. Capital letters such as  $A, B, C, \ldots$  will be used to denote nonterminals, small letters such as  $a, b, c, \ldots$  will be used to denote terminals, and greek letters such as  $\alpha, \beta, \ldots$  will be used to denote any string from  $(N \cup T)^*$ .

A production of the form  $A \to \epsilon$  is called an  $\epsilon$ -production. A production of the kind  $A \to B$  is known as a unit production.

Let G be a CFG such that L(G) does not have  $\epsilon$ . Then we can convert G into Chomsky Normal Form (CNF). A context free grammar is in CNF if the productions in P are of only two kinds:  $A \to a$  and  $A \to BC$ .

Let U and V be two  $n \times n$  real matrices. Then the tropical (or distance) product Z of X and Y is defined as:  $Z_{ij} = \min_{k=1}^{n} (X_{ik} + Y_{kj}), \ 1 \leq i, j \leq n$ .

The edit distance between two strings  $\mathcal{I}$  and  $\mathcal{I}'$  from an alphabet T is the minimum number of (insert, delete, and substitution) operations needed to convert  $\mathcal{I}$  to  $\mathcal{I}'$ .

In this paper we assume that the grammar size is O(1) which is a standard assumption made in many works (see e.g., [13]).

### 1.2 Some Preliminaries

#### 1.2.1 A summary of Aho and Peterson's algorithm

The algorithm of Aho and Peterson [1] is based on the parsing algorithm of Earley [4]. There are some crucial differences. Let  $\mathcal{I} = a_1 a_2 \dots a_n$  be the input string. If G(N, T, P, S) is the input grammar, another grammar G' = (N', T, P', S') is constructed where G' has all the productions of G and some additional productions that can be used to make derivations involving errors. Each such additional production is called an *error production*. Three kinds of errors are considered, namely, insertion, deletion, and substitution. G' also has some additional nonterminals. The algorithm derives the input string beginning with S', minimizing the number of applications of the error productions.

The parser of [1] can be thought of as a modified version of the Earley parser. Like the algorithm of Earley, n+1 levels of lists are constructed. Each list consists of items where an item is an object of the form  $[A \to \alpha.\beta, i, k]$ . Here  $A \to \alpha\beta$  is a production, . is a special symbol that indicates what part of the production has been processed so far, i is an integer indicating input position at which the derivation of  $\alpha$  started, and k is an integer indicating the number of error productions that have been used in the derivation from  $\alpha$ . If we use  $\mathcal{L}_j$  to denote the list of level  $j, 0 \le i \le n$ , then the item  $[A \to \alpha.\beta, i, k]$  will be in  $\mathcal{L}_j$  if and only if for some  $\nu$  in  $(N \cup T)^*$ ,  $S' \stackrel{*}{\Rightarrow} a_1 a_2 \cdots a_i A \nu$  and  $\alpha \stackrel{*}{\Rightarrow} a_{i+1} a_{i+2} \cdots a_j$  using k error productions.

The algorithm constructs the lists  $\mathcal{L}_0, \mathcal{L}_1, \dots, \mathcal{L}_n$ . An item of the form  $[S' \to \alpha, 0, k]$  will be in  $\mathcal{L}_n$ , for some integer k. In this case, k is the minimum edit distance between  $\mathcal{I}$  and any string in L(G).

Note that the Earley parser also works in the same manner except that an item will only have two elements:  $[A \to \alpha.\beta, i]$ .

#### 1.2.2 A synopsis of Valiant's algorithm

Valiant has presented an efficient algorithm for computing the transitive closure of an upper triangular matrix. The transitive closure is with respect to matrix multiplication defined in a specific way. Each element in a matrix will be a set of items. In the case of context free recognition, each matrix element will be a set of nonterminals. If  $N_1$  and  $N_2$  are two sets of nonterminals, a binary operator  $\cdot$  is defined as:  $N_1 \cdot N_2 = \{A | \exists B \in N_1, C \in N_2 \text{ such that } (A \to BC) \in P\}$ . If a and b are matrices where each element is a subset of N, the product matrix c of a and b is defined as follows.

$$c_{ij} = \bigcup_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Under the above definition of matrix multiplication, we can define transitive closure for any matrix a as:

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

where  $a^{(1)} = a$  and

$$a^{(i)} = \bigcup_{j=1}^{i-1} a^{(j)} \cdot a^{(i-j)}$$

Valiant has shown that this transitive closure can be computed in O(S(n)) time, where S(n) is the time needed for multiplying two matrices with the above special definition of matrix product. In fact this algorithm works for the computation of transitive closure for generic operators  $\odot$  and union as long as these operations satisfy the following properties: The outer operation (i.e., union) is commutative and associative, the inner operation  $(\odot)$  distributes over union,  $\emptyset$  is a multiplicative zero and an additive identity.

# 2 A Simple Error Correcting Parser

In this section we present a simple error correcting parser for CFGs. This algorithm is based on the algorithm of [10]. We also utilize the concept of error productions introduced in [1]. If G = (N, T, P, S) is the input grammar, we generate another grammar G' = (N', T, P', S) where  $N' = N \cup \{H, I\}$ . P' has all the productions in P. In addition, P' has some additional error productions. We parse the given input string  $\mathcal{I}$  using the productions in P'. For each production, we associate an error count that indicates the minimum number of errors the use of the production will amount to. The goal is to parse  $\mathcal{I}$  using as few error productions as possible. Specifically, the sum of error counts of all the error productions used should be minimum. If  $A \to \alpha$  is an error production with an error count of k, we denote this rule as  $A \stackrel{k}{\to} \alpha$ . If there is no integer above  $\to$  in any production, the error count of this production should be assumed to be 0.

## 2.1 Construction of a covering grammar

Let G = (N, T, P, S) be the given grammar and  $\mathcal{I} = a_1 a_2 \dots a_n$  be the given input string. Without loss of generality assume that L(G) does not have  $\epsilon$  and that G is in CNF. Even if G is not in CNF, we could employ standard techniques to convert G into this form (see e.g., [5]). We construct a new grammar G' = (N', T, P', S) as follows. P' has the following productions in addition to the ones in  $P: H \xrightarrow{0} HI, H \xrightarrow{0} I$ , and  $I \xrightarrow{1} a$  for every  $a \in T$ . Here H and I are new nonterminals. If  $A \to a$  is in P, then add the following rules to P':

 $A \xrightarrow{1} b$  for every  $b \in T - \{a\}$ ,  $A \xrightarrow{1} \epsilon$ ,  $A \xrightarrow{0} AH$ , and  $A \xrightarrow{0} HA$ . Each production in P has an error count of 0.

Elimination of  $\epsilon$ -productions: We first eliminate the  $\epsilon$ -productions in P' as follows. We say a nonterminal A is nullable if  $A \stackrel{*}{\Rightarrow} \epsilon$ . Let k be the number of errors needed for A to derive  $\epsilon$ . We denote this as follows:  $A \stackrel{*,k}{\Rightarrow} \epsilon$ . Call k the nullcount of A, denoted as nullcount A. We only keep the minimum such nullcount for any nonterminal. Let the minimum nullcount for any terminal A be Mnullcount(A). For example, if  $A \stackrel{0}{\rightarrow} BC, B \stackrel{1}{\rightarrow} \epsilon$ , and  $C \stackrel{1}{\rightarrow} \epsilon$  are in P', then  $A \stackrel{*,2}{\Rightarrow} \epsilon$ . We identify all the nullable nonterminals in P' using the following procedure. If  $B \rightarrow CD$  is in P' and if both C and D are nullable, then B is nullable as well. In this case, nullcount(B) = nullcount(C) + nullcount(D).

After identifying all nullable nonterminals and their Mnullcount values, we process each production as follows. Let  $A \stackrel{k}{\to} BC$  be any production in P'. If B is nullable and C is not, and if  $Mnullcount(B) = \ell$ , then we add the production  $A \stackrel{k+\ell}{\to} C$  to P'. If C is nullable and B is not, and if  $Mnullcount(C) = \ell$ , then we add the production  $A \stackrel{k+\ell}{\to} B$  to P'. If both or none of B and C are nullable, then we do not add any additional production to P' while processing the production  $A \stackrel{k}{\to} BC$ . If there are more than one productions in P' with the same precedent and consequent, we only keep that production for which the error count is the least.

Finally, we remove all the  $\epsilon$  productions.

Elimination of unit productions: We eliminate unit productions from P' as follows. Let  $A \stackrel{k_1}{\to} B_1, B_1 \stackrel{k_2}{\to} B_2, \dots, B_{q-3} \stackrel{k_{q-2}}{\to} B_{q-2}, B_{q-2} \stackrel{k_{q-1}}{\to} B$  be a sequence of unit productions in P' and  $B \stackrel{k_q}{\to} \alpha$  be a non unit production. In this case we add the production  $A \stackrel{Q}{\to} \alpha$  to P', where  $Q = \sum_{i=1}^q k_i$ . After processing all such sequences and adding productions to P' we eliminate duplicates. In particular, if there are more than one rules with the same precedent and consequent, we only keep the production with the least error count. At the end we remove all the unit productions.

**Observation:** Also and Peterson [1] indicate that G' is a covering grammar for G and prove several properties of G'. Note that they don't keep any error counts with their productions. Also, the validity of the procedures we have used to eliminate  $\epsilon$  and unit productions can be found in [5].

**An Example.** Consider the language  $\{a^nb^n : n \geq 1\}$ . A CFG for this language has the productions:  $S \to aSb|ab$ . We can get an equivalent grammar G = (N, T, P, S) in CNF

where  $N = \{S, A, B, A_1\}, T = \{a, b\}$ , and  $P = \{S \to AA_1 | AB, A_1 \to SB, A \to a, B \to b\}$ . We can get a grammar  $G_1 = (N', T, P_1, S)$  with error productions where  $N' = \{S, A, B, A_1, H, I\}$  and  $P_1 = \{S \to AA_1 | AB, A_1 \to SB, A \to a, B \to b, H \to HI, H \to I, I \xrightarrow{1} a, I \xrightarrow{1} b, A \xrightarrow{1} b, A \xrightarrow{1} \epsilon, A \to HA, A \to AH, B \xrightarrow{1} a, B \xrightarrow{1} \epsilon, B \to HB, B \to BH\}$ . Note that any production with no integer above  $\to$  has an error count of zero.

- Eliminating  $\epsilon$ -productions: We identify nullable nonterminals. We realize that the following nonterminals are nullable: A, B, S, and  $A_1$ . For example,  $A_1$  is nullable since we have:  $A_1 \stackrel{*}{\Rightarrow} SB \stackrel{*}{\Rightarrow} ABB \stackrel{*,1}{\Rightarrow} BB \stackrel{*,1}{\Rightarrow} \epsilon$ . We also realize:  $A \stackrel{*,1}{\Rightarrow} \epsilon, B \stackrel{*,1}{\Rightarrow} \epsilon, S \stackrel{*,2}{\Rightarrow} \epsilon$ , and  $A_1 \stackrel{*,3}{\Rightarrow} \epsilon$ .
  - Now we process every production in  $P_1$  and generate new relevant rules. For instance, consider the rule  $S \to AA_1$ . Since  $A_1 \stackrel{*,3}{\Rightarrow} \epsilon$ , we add the rule  $S \stackrel{3}{\to} A$  to  $P_1$ . When we process the rule  $S \to AB$ , since  $B \stackrel{*,1}{\Rightarrow} \epsilon$ , we realize that  $S \stackrel{1}{\to} A$  has to be added to  $P_1$ . However,  $S \stackrel{3}{\to} A$  has already been added to  $P_1$ . Thus we replace  $S \stackrel{3}{\to} A$  with  $S \stackrel{1}{\to} A$ . Processing in a similar manner, we add the following productions to  $P_1$  to get  $P_2$ :  $S \stackrel{1}{\to} A_1, S \stackrel{1}{\to} A, S \stackrel{1}{\to} B, A_1 \stackrel{1}{\to} S, A_1 \stackrel{2}{\to} B, A \stackrel{1}{\to} H$ , and  $B \stackrel{1}{\to} H$ . We eliminate all the  $\epsilon$ -productions from  $P_2$ .
- Eliminating unit productions: We consider every sequence of unit productions  $A \stackrel{k_1}{\to} B_1, B_1 \stackrel{k_2}{\to} B_2, \dots, B_{q-3} \stackrel{k_{q-2}}{\to} B_{q-2}, B_{q-2} \stackrel{k_{q-1}}{\to} B$  in  $P_2$  with  $B \stackrel{k_q}{\to} \alpha$  being a non unit production. In this case we add the production  $A \stackrel{Q}{\to} \alpha$  to  $P_2$ , where  $Q = \sum_{i=1}^q k_i$ . Consider the sequence  $S \stackrel{1}{\to} A, A \to a$ . This sequence results in a new production:  $S \stackrel{1}{\to} a$ . The sequence  $S \stackrel{1}{\to} A_1, A_1 \stackrel{2}{\to} B, B \stackrel{1}{\to} a$  suggests the addition of the production  $S \stackrel{4}{\to} a$ . But we have already added a better production and hence this production is ignored.

Proceeding in a similar manner we realize that we have to add the following productions to  $P_2$  to get  $P_3$ :  $S \xrightarrow{1} a, S \xrightarrow{1} b, H \xrightarrow{1} a, H \xrightarrow{1} b, A_1 \xrightarrow{2} a, A_1 \xrightarrow{2} b, S \xrightarrow{1} HB, S \xrightarrow{1} BH, A \xrightarrow{1} HI, B \xrightarrow{1} HI, S \xrightarrow{2} HI, A_1 \xrightarrow{3} HI, A_1 \xrightarrow{2} BH, A_1 \xrightarrow{2} HB, S \xrightarrow{1} AH, S \xrightarrow{1} HA, A_1 \xrightarrow{2} AH$ , and  $A_1 \xrightarrow{2} HA$ . We eliminate all the unit productions from  $P_3$ .

The final grammar we get is  $G_3 = (N', T, P_3, S)$  where  $P_3 = \{S \rightarrow AA_1 | AB, A_1 \rightarrow SB, A \rightarrow a, B \rightarrow b, H \rightarrow HI, I \xrightarrow{1} a, I \xrightarrow{1} b, A \xrightarrow{1} b, A \rightarrow HA, A \rightarrow AH, B \xrightarrow{1} a, B \rightarrow HB, B \rightarrow BH, S \xrightarrow{1} a, S \xrightarrow{1} b, H \xrightarrow{1} a, H \xrightarrow{1} b, A_1 \xrightarrow{2} a, A_1 \xrightarrow{2} b, S \xrightarrow{1} HB, S \xrightarrow{1} BH, A \xrightarrow{1} HI, B \xrightarrow{1} HI, S \xrightarrow{2} HI, A_1 \xrightarrow{3} HI, A_1 \xrightarrow{2} BH, A_1 \xrightarrow{2} HB, S \xrightarrow{1} AH, S \xrightarrow{1} HA, A_1 \xrightarrow{2} AH, A_1 \xrightarrow{2} HA\}.$ 

## 2.2 The algorithm

The algorithm is a modified version of an algorithm given in [10]. This algorithm in turn is a slightly different version of the CYK algorithm. Let G' = (N', T', P', S') be the grammar generated using the procedure given in Section 2.1. The basic idea behind the algorithm is the following: The algorithm has n stages. In any given stage we scan through each production in P' and grow larger and larger parse trees. At any given time in the algorithm, each nonterminal has a list of tuples of the form  $(i, j, \ell)$ . If A is any nonterminal, LIST(A) will have tuples  $(i, j, \ell)$  such that  $A \stackrel{*,\ell}{\Rightarrow} a_i \dots a_{j-1}$  and there is no  $\ell' < \ell$  such that  $A \stackrel{*,\ell'}{\Rightarrow} a_i \dots a_{j-1}$ . If  $A \stackrel{\ell_3}{\Rightarrow} BC$  is a production in P', then in any stage we process this production as follows: We scan through elements in LIST(B) and look for matches in LIST(C). For instance, if  $(i, k, \ell_1)$  is in LIST(B) (for some integer  $\ell_1$ ), we check if  $(k, j, \ell_2)$  is in LIST(C), for some j and  $\ell_2$ . If so, we insert  $(i, j, \ell_1 + \ell_2 + \ell_3)$  into LIST(A). If LIST(A) for any A has many tuples of the form (i, j, \*) we keep only one among these. Specifically, if  $(i, j, \ell_1), (i, j, \ell_2), \dots, (i, j, \ell_q)$  are in LIST(A), we keep only  $(i, j, \ell)$  where  $\ell = \min_{m=1}^q \ell_m$ .

We maintain the following data structures: (1) for each nonterminal A, an array (call it  $X_A$ ) of lists indexed 1 through n, where  $X_A[i]$  is the list of all tuples from LIST(A) whose first item is i ( $1 \le i \le n$ ); and (2) an  $n \times n$  upper triangular matrix  $\mathcal{M}$  whose (i, j)th entry will be those nonterminals that derive  $a_i a_{i+1} \dots a_{j-1}$  with the corresponding (minimum) error counts (for  $1 \le i \le n$  and  $2 \le j \le (n+1)$ ). There can be  $O(n^2)$  entries in LIST(B), and for each entry (i, k) in this list, we need to search for at most n items in LIST(C).

By induction, we can show that at the end of stage s  $(1 \le s \le n)$ , the algorithm would have computed all the nonterminals that span any input segment of length s or less, together with the minimum error counts. (We say a nonterminal spans the input segment  $J = a_i a_{i+1} \dots a_{j-1}$  if it derives J; the nonterminal is said to have a "span-length" of j - i.)

A straight forward implementation of the above idea takes  $O(n^4)$  time. However, we can reduce the run time of each stage to  $O(n^2)$  as follows: In stage s, while processing the production  $A \to BC$ , work only with tuples from LIST(B) and LIST(C) whose combination will derive an input segment of length exactly s. For example, if  $(i, k, \ell)$  is a tuple in LIST(B), the only tuple in C we should look for is  $(k, i+s, \ell')$  (for any integer  $\ell'$ ). We can look for such a tuple in O(1) time using the matrix  $\mathcal{M}$ . With this modification, each stage of the above algorithm will only take  $O(n^2)$  time and hence the total run time of the algorithm is  $O(n^3)$ . The pseudocode is given in algorithm 1.

**Theorem 2.1** When the above algorithm completes, for any nonterminal A, LIST(A) has a tuple  $(i, j, \ell)$  if and only if  $A \stackrel{*, \ell}{\Rightarrow} a_i \dots a_{j-1}$  and there is no  $\ell' < \ell$  such that  $A \stackrel{*, \ell'}{\Rightarrow} a_i \dots a_{j-1}$ .

**Proof:** The proof is by induction on the stage number s.

```
Algorithm 1: ErrorCorrectingParser(G, \mathcal{I})
 input : G = (N, T, P, S), a grammar;
                \mathcal{I} = a_1 a_2 \dots a_n, input string;
  output: minimum distance \ell between \mathcal{I} and any string in L(G);
  begin
      Generate G' using the procedure in Section 2.1;
      for A \in N' and i \leftarrow 1 to n do
        X_A[i] := \{\};
      for i \leftarrow 1 to n and j \leftarrow (i+1) to (n+1) do
        \mathcal{M}_{i,j} = \{\};
      for i \leftarrow 1 to n do
           for (A \stackrel{\ell}{\rightarrow} a_i) \in P' do
                insert (A, \ell) into \mathcal{M}_{i,i+1};
                insert (i, i+1, \ell) into X_A[i];
      for s \leftarrow 2 to n do
           for (A \stackrel{\ell_3}{\rightarrow} BC) \in P' do
                for (i, k, \ell_1) \in LIST(B) do
                     for (C, \ell_2) \in \mathcal{M}_{k,i+s} do
                         \ell := \ell_1 + \ell_2 + \ell_3;
                         insert (A, \ell) into \mathcal{M}_{i,i+s};
                         insert (i, i + s, \ell) into X_A[i];
      find \ell for which (1, n + 1, \ell) \in X_S[1];
      return \ell;
```

Figure 1: Algorithm 1.

**Base Case:** is when s=1, i.e., j-i=1, for  $1 \le i, j \le (n+1)$ . Note that all the nonterminals other than H and I are nullable. As a result, P' will have a production of the kind  $A \xrightarrow{\ell} b$  for every nonterminal A and every terminal b, for some integer  $\ell$ . By the way we compute Mnullcount for each nonterminal and eliminate unit productions, it is clear that  $\ell$  is the smallest integer for which  $A \stackrel{*,\ell}{\Rightarrow} b$ .

**Induction Step:** Assume that the hypothesis is true for span lengths up to s-1. We can prove it for a span length of s. Let  $A \stackrel{l_3}{\to} BC$  be any production in P'. Let  $(i, k, \ell_1)$  be a tuple in LIST(B) and  $(k, j, \ell_2)$  be a tuple in LIST(C) with j-i=s. Then this means that  $A \stackrel{*,L}{\to} a_i \dots a_{j-1}$ , where  $L = \ell_1 + \ell_2 + \ell_2$ . We add the tuple (i, j, L) to LIST(A). Also, the induction hypothesis implies that  $B \stackrel{*,\ell_1}{\to} a_i \dots a_{k-1}$  and there is no  $\ell < \ell_1$  for which

 $B \stackrel{*,\ell}{\Rightarrow} a_i \dots a_{k-1}$ . Likewise,  $\ell_2$  is the smallest integer for which  $C \stackrel{*,\ell_2}{\Rightarrow} a_k \dots a_{j-1}$ . We consider all such productions in P that will contribute tuples of the kind (i,j,\*) to LIST(A) and from these only keep  $(i,j,\ell)$  where  $\ell$  is the least such integer.  $\square$ 

## 3 Less than Cubic Time Parser

In this section we present an error correcting parser that runs in time O(T(n)) where n is the length of the input string and T(n) is the time needed to compute the tropical product of two  $n \times n$  matrices. There are two main ingredients in this parser, namely, the procedure given in Section 2.1 for converting the given grammar into a covering grammar and Valiant's reduction given in [13] (and summarized in Section 1.2.2).

As pointed out in Section 1.2.2, Valiant has presented an efficient algorithm for computing the transitive closure of an upper triangular matrix. The transitive closure is with respect to matrix multiplication defined in a special way. Each element in a matrix will be a set of items. In standard matrix multiplication we have two operators, namely, multiplication and addition. In the case of special matrix multiplication, these operations are replaced by  $\odot$  and  $\cup$  (called *union*). Valiant's algorithm works as long as these operations satisfy the following properties: The outer operation (i.e., union) is commutative and associative, the inner operation  $\odot$  distributes over union,  $\emptyset$  is a zero with respect to  $\odot$  and an identity with respect to union.

Valiant has shown that transitive closure under the above definition of matrix multiplication can be computed in O(S(n)) time, where S(n) is the time needed for multiplying two matrices with the above special definition of matrix product.

In the context of error correcting parser we define the two operations as follows. Let  $\mathcal{I} = w_1 w_2 \dots w_n$  be the input string. Matrix elements are sets of pairs of the kind  $(A, \ell)$  where A is a nonterminal and  $\ell$  is an integer. We initialize an  $(n+1) \times (n+1)$  upper triangular matrix a as:

$$a_{i,i+1} = \{(A,\ell) | (A \xrightarrow{\ell} w_i) \in P'\}, \text{ and } a_{i,j} = \emptyset, \text{ for } j \neq i+1.$$
 (1)

If  $N_1$  and  $N_2$  are sets of pairs of the kind  $(A, \ell)$  then  $N_1 \odot N_2$  is defined with the procedure in algorithm 2.

If  $N_1$  and  $N_2$  are sets of pairs of the kind  $(A, \ell)$ , the union operation is defined as:

$$N_1 \cup N_2 = \{(A,k): (1) \ (A,k) \in N_1 \text{ and } (A,k') \notin N_2 \text{ for any } k' \text{ or } (2) \ (A,k) \in N_2 \text{ and } (A,k') \notin N_2 \text{ for any } k' \text{ or } (2) \text{ or } (3) \text{ or }$$

## Algorithm 2: $\odot(N_1, N_2)$

Figure 2: Algorithm 2.

$$(A, k') \notin N_1$$
 for any  $k'$  or  $(3)$   $(A, k_1) \in N_1$ ,  $(A, k_2) \in N_2$  and  $k = \min(k_1, k_2)$ .

It is easy to see that the union operation is commutative and associative since if there are multiple pairs for the same nonterminal with different error counts, the union operation has the effect of keeping only one pair for each nonterminal with the least error count.

We can also verify that  $\odot$  distributes over union. Let  $N_1, N_2$ , and  $N_3$  be any three sets of pairs of the type  $(A, \ell)$ . Let  $Q = N_1 \odot (N_2 \cup N_3), R = (N_1 \odot N_2) \cup (N_1 \odot N_3), X = (N_1 \odot N_2),$  and  $Y = (N_1 \odot N_3)$ . If  $(B, \ell) \in N_1, (C, k_1) \in N_2, (C, k_2) \in N_3, k = \min(k_1, k_2),$  and  $(A \xrightarrow{m} BC) \in P'$ , then  $(A, k+\ell+m) \in Q$ . Also,  $(A, k_1+\ell+m) \in X$  and  $(A, k_2+\ell+m) \in Y$ . Thus,  $(A, k+\ell+m) \in R$ .

Put together, we get the following algorithm. Given a grammar G and an input string  $\mathcal{I}$ , generate the grammar G' using the procedure in Section 2.1. Construct the matrix a described in Equation 1. Compute the transitive closure  $a^+$  of a using Valiant's algorithm [13].  $(S, \ell)$  will occur in  $a_{1,n+1}^+$  for some integer  $\ell$ . In this case, the minimum distance between  $\mathcal{I}$  and any string in L(G) is  $\ell$ . The pseudocode is given in algorithm 3.

Note that, by definition,

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

where  $a^{(1)} = a$  and

$$a^{(i)} = \bigcup_{j=1}^{i-1} a^{(j)} \cdot a^{(i-j)}.$$

It is easy to see that  $(A, \ell)$ , where A is a nonterminal and  $\ell$  is an integer in the range [0, n], will be in  $a_{i,j}^{(k)}$  if and only if  $A \stackrel{*,\ell}{\Rightarrow} a_i a_{i+1} \cdots a_{j-1}$  such that (j-i) = k and there is no

#### Algorithm 3: ErrorCorrectingParser2( $G, \mathcal{I}$ )

```
input : G, a grammar; \mathcal{I} = w_1w_2 \dots w_n, a string; output: the minimum distance \ell between \mathcal{I} and any string in L(G); begin

Generate G' using the procedure in Section 2.1; for i \leftarrow 1 to n and j \leftarrow (i+1) to (n+1) do

a_{i,j} := \{\}; for i := 1 to n do

a_{i,i+1} := \{(A,\ell)|(A \xrightarrow{\ell} w_i) \in P'\}; a^+ := \text{TransitiveClosure } (a); // \text{ using Valiant's algorithm } [13] return \ell for which (S,\ell) \in a_{1,n+1}^+;
```

Figure 3: Algorithm 3.

```
q < \ell such that A \stackrel{*,q}{\Rightarrow} a_i a_{i+1} \cdots a_{j-1}. Also, a_{i,j}^{(k)} = \emptyset if (j-i) \neq k.
```

As a result,  $(S, \ell)$  will be in  $a_{1,n+1}^+$  if and only if  $S \stackrel{*,\ell}{\Rightarrow} a_1 a_2 \cdots a_n$  and there is no  $q < \ell$  such that  $S \stackrel{*,q}{\Rightarrow} a_1 a_2 \cdots a_n$ .

We get the following

**Theorem 3.1** Error correcting parsing can be done in O(S(n)) time where S(n) is the time needed to multiply two matrices under the new definition of matrix product.  $\square$ 

It remains to be shown that S(n) = O(T(n)) where T(n) is the time needed to compute the tropical product of two matrices.

Let a and b be two matrices where the matrix elements are sets of pairs of the kind  $(A, \ell)$  where A is a nonterminal and  $\ell$  is an integer. Let c = ab be the product of interest under the special definition of matrix product.

For each nonterminal B in G', we define a matrix  $a_B$  and for each nonterminal C in G', we define a matrix  $b_C$ .  $a_B[i,j] = \ell$  if  $(B,\ell) \in a[i,j]$ , for  $1 \le i,j \le n$ . Likewise,  $b_C[i,j] = \ell$  if  $(C,\ell) \in b[i,j]$ , for  $1 \le i,j \le n$ . The pseudocode of this operation is given in algorithm 4. Compute the tropical product  $c_{BC}$  of  $a_B$  and  $b_C$  for every nonterminal B and every nonterminal C.

For every production  $A \stackrel{k}{\to} BC$  in P' do the following: for every  $1 \leq i, j \leq n$ , if

 $c_{BC}[i,j] = \ell$  then add  $(A, k + \ell)$  to c[i,j], keeping only the smallest distance if A is already present in c[i,j]. The pseudocode is given in algorithm 5.

```
      Algorithm 4: DistMatrix(a, B)

      input : a, m \times n matrix where each entry is a set of pairs (A, \ell) such that A is a nonterminal and \ell is an integer;

      B: a nonterminal;

      output: a_B, m \times n matrix where each entry is an integer \ell such that either (B, \ell) \in a[i, j] or \ell = +\infty;

      begin

      for i \leftarrow 1 to m and j \leftarrow 1 to n do

      if \exists \ell for which (B, \ell) \in a[i, j] then

      a_B[i, j] := \ell;

      else

      a_B[i, j] := +\infty;

      return a_B;
```

Figure 4: Algorithm 4.

Clearly, the time spent in computing ab (under the new special definition of matrix product) is  $O(T(n) + n^2)$  assuming that the size of the grammar is O(1).

Put together we get the following theorem.

**Theorem 3.2** Error correcting parsing can be done in O(T(n)) time where T(n) is the time needed to compute the tropical product of two matrices.  $\square$ 

Using the currently best known algorithm for tropical products [2], we get the following theorem.

**Theorem 3.3** Error correcting parsing can be done in 
$$O\left(\frac{n^3(\log\log n)^3}{(\log n)^2}\right)$$
 time.  $\square$ 

Furthermore, consider the case of error correcting parsing where we know a priori that there exists a string  $\mathcal{I}'$  in L(G) such that the distance between  $\mathcal{I}$  and  $\mathcal{I}'$  is upper bounded by m. We can solve this version of the language edit distance problem using the tropical matrix product algorithm of Zwick [15]. This algorithm multiplies two  $n \times n$  integer matrices in  $O(Mn^{\omega})$  time if the matrix elements are in the range [-M, M] [15]. Here  $O(n^{\omega})$  is the

# Algorithm 5: MatrixMultiplication(a, b)input : a,b: $m \times q$ and $q \times n$ matrices respectively; each entry of a and b is a set of pairs $(A, \ell)$ such that A is a nonterminal and $\ell$ is an integer; **output**: c = ab where multiplication is done with respect to $(\odot, \cup)$ ; begin for $i \leftarrow 1$ to m and $j \leftarrow 1$ to n and $A \in N'$ do $\min \operatorname{Dist}_{i,j}[A] := +\infty;$ for $B \in N'$ do $a_B := DistMatrix (a, B);$ for $C \in N'$ do $b_C := DistMatrix (b, C);$ $c_{BC} := TropicalProduct (a_B, b_C);$ for $(A \stackrel{k}{\rightarrow} BC) \in P'$ do for $i \leftarrow 1$ to m and $j \leftarrow 1$ to n do $\ell := k + c_{BC}[i, j];$ $\min Dist_{i,j}[A] := \min(\ell, \min Dist_{i,j}[A]);$ for $i \leftarrow 1$ to m and $j \leftarrow 1$ to n do $c[i,j] := \{(A,\ell)|A \in N', \min \text{Dist}_{i,j}[A] = \ell, \ell \neq \infty\};$ return c;

Figure 5: Algorithm 5.

time taken to multiply two  $n \times n$  real matrices. Recall that when we reduce the matrix multiplication under  $\odot$  and  $\cup$  to tropical matrix multiplication, we have to compute the tropical product  $c_{BC}$  of  $a_B$  and  $b_C$  for every nonterminal B and every nonterminal C. Elements of  $a_B$  and  $b_C$  are integers in the range [0, n]. Note that even if all the elements of  $c_{BC}$  are  $\leq m$ , some of the elements of  $a_B$  and  $b_C$  could be larger than m. Before using the algorithm of [15] we have to ensure that all the elements of  $a_B$  and  $b_C$  are less than M (where M is some function of m). This can be done as follows. Before invoking the algorithm of [15], we replace every element of  $a_B$  and  $b_C$  by m+1 if the element is > m. m+1 is 'infinity' as far as this multiplication is concerned. By doing this replacement, we are not affecting the final result of the algorithm and at the same time, we are making sure

that the elements of  $a_B$  and  $b_C$  are  $\leq M = (m+1)$ .

As a result, we get the following theorem.

**Theorem 3.4** Error correcting parsing can be done in  $O(mn^{\omega})$  time where m is an upper bound on the edit distance between the input string  $\mathcal{I}$  and some string  $\mathcal{I}'$  in L(G), G being the input CFG.  $O(n^{\omega})$  is the time it takes to multiply two  $n \times n$  matrices.  $\square$ 

As a corollary to the above theorem we can also get the following theorem.

**Theorem 3.5** There exists an  $\frac{n}{m}$ -approximation algorithm for the language edit distance problem that has a run time of  $O(mn^{\omega})$ , where  $O(n^{\omega})$  is the time taken to multiply two  $n \times n$  matrices.

**Proof:** Here again, we replace every element of  $a_B$  and  $b_C$  by m+1 if the element is > m. In this case the elements of  $c_{BC}$  will be  $\leq (2m+2)$ . We replace any element in  $c_{BC}$  that is larger than m with (m+1). In general whenever we generate or operate on a matrix, we will ensure that the elements are  $\leq (m+1)$ . If  $S \stackrel{\ell}{\Rightarrow} \mathcal{I}$  for some  $\ell \leq m$ , then the final answer output will be exact. If  $\ell > m$ , then the algorithm will always output n. Thus the theorem follows.  $\square$ 

# 4 Retrieving $\mathcal{I}'$

In all the algorithms presented above, we have focused on computing the minimum edit distance between the input string  $\mathcal{I}$  and any string  $\mathcal{I}'$  in L(G). In this section we address the problem of finding  $\mathcal{I}'$ . We show that  $\mathcal{I}'$  can be found in  $O(n^2)$  time, where  $n = |\mathcal{I}|$ . Let  $S \stackrel{*,\ell}{\Rightarrow} \mathcal{I}$  such that there is no  $k < \ell$  such that  $S \stackrel{*,k}{\Rightarrow} \mathcal{I}$ . Let  $\mathcal{I} = a_1 a_2 \cdots a_n$ .

Realize that in the algorithms given in Section 2.2 and Section 3 we compute, for every i and j (with j > i), all the nonterminals A such that A spans  $a_i a_{i+1} \dots a_{j-1}$  and we also determine the least k such that  $A \stackrel{*,k}{\Rightarrow} a_i a_{i+1} \dots a_{j-1}$ . In this case, there will be an entry for A in the matrix  $\mathcal{M}$ . Specifically, (A, k) will be in  $\mathcal{M}(i, j)$ . We can utilize this information to identify an  $\mathcal{I}'$  such that the edit distance between  $\mathcal{I}$  and  $\mathcal{I}'$  is equal to  $\ell$ . Note that we can deduce  $\mathcal{I}'$  if we know the sequence of productions used to derive  $\mathcal{I}'$ . The pseudocode is given in algorithm 6. We will invoke the algorithm as  $ParseTree(\mathcal{M}, S, 1, n+1, \ell)$ .

Algorithm 6 finds the first production in O(n) time. Having found the first production, we can proceed in a similar manner to find the other productions needed to derive  $\mathcal{I}'$ . In the second stage we have to find a production that can be used to derive  $a_1a_2...a_j$  from A and another production that can be used to derive  $a_{j+1}a_{j+2}...a_n$  from B. Note that

```
Algorithm 6: ParseTree(M, D, i, j, \ell)
 input : M, transitive closure matrix as discussed in the text;
               D, a non terminal;
               i, j-1, start and end position in the input string \mathcal{I}, where \mathcal{I} = a_1 a_2 \dots a_n;
               e, an edit distance;
  output: a parse tree which can derive \mathcal{I}[i..(j-1)] from D with \ell errors;
  begin
      if i = j - 1 then
           find \ell for which (D, \ell) \in \mathcal{M}_{i,j};
          return new node(i, j, D \xrightarrow{\ell} a_i);
      for k \leftarrow (i+1) to (j-1) do
          if \exists (A, q_1) \in \mathcal{M}(i, k) and \exists (B, q_2) \in \mathcal{M}(k, j) then
               if \exists (D \stackrel{q_3}{\rightarrow} AB) \in P' and q_1 + q_2 + q_3 = \ell then
                    break;
      T_1 := ParseTree(\mathcal{M}, A, i, k, q_1);
      T_2 := ParseTree(\mathcal{M}, B, k, j, q_2);
      r := \text{new node}(i, j, D \stackrel{q_3}{\rightarrow} AB);
      left(r) := T_1;
      right(r) := T_2;
      return r;
```

Figure 6: Algorithm 6.

the span length of A plus the span length of B is n and hence both the productions can be found in a total of O(n) time.

We can think of a tree  $\mathcal{T}$  where S is the root and S has two children A and B. If  $A \to CD$  is the first production that can be used to derive  $a_1 a_2 \dots a_j$  from A and  $B \to EF$  is the first production that can be used to derive  $a_{j+1} a_{j+2} \dots a_n$  from B, then A will have two children C and D and B will have two children E and F.

The rest of the tree is constructed in the same way. Clearly, the total span length of all the nonterminals in any level of the tree is  $\leq n$  and hence the time spent at each level is O(n). Also, there can be at most n levels. As a result, we get the following theorem.

**Theorem 4.1** We can identify  $\mathcal{I}'$  in  $O(n^2)$  time.  $\square$ 

## 5 Conclusions

In this paper we have presented an error correcting parser for general context free languages. This algorithm takes less than cubic time, improving the 1972 algorithm of Aho and Peterson that has remained the best until now. We have also shown that if M is an upper bound on the edit distance between the input string  $\mathcal{I}$  and some string of L(G), then we can solve the parsing problem in  $O(Mn^{\omega})$  time, where  $O(n^{\omega})$  is the time it takes to multiply two  $n \times n$  matrices. As a corollary, we have presented an  $\frac{n}{M}$ -approximation algorithm for the general context free language edit distance problem that runs in time  $O(Mn^{\omega})$ .

# Acknowledgements

This work has been supported in part by the following grant: NIH R01LM010101. The first author thanks Barna Saha for the introduction of this problem and Alex Russell for providing pointers to tropical matrix multiplication.

## References

- [1] A.V. Aho and T.G. Peterson, A minimum distance error-correcting parser for context-free languages, SIAM Journal on Computing 1(4), 1972, 305-312.
- [2] T.M. Chan, More algorithms for all-pairs shortest paths in weighted graphs, SIAM Journal on Computing 39(5), 2010, 2075-2089.
- [3] J. Cocke and J.T. Schwartz, Programming languages and their compilers: Preliminary notes, Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.
- [4] J. Earley, An efficient context-free parsing algorithm, Communications of the ACM 13, 1970, 94-102.
- [5] J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Third Edition, Prentice Hall, July 9, 2006.
- [6] J. Kasami, An efficient recognition and syntax analysis algorithm for context-free languages, Report of Univ. of Hawaii, 1965.
- [7] F. Korn, B. Saha, D. Srivastava, and S. Ying, On repairing structural problems in semi-structured data, *Proc. VLDB*, 2013.

- [8] L. Lee, Fast context-free grammar parsing requires fast boolean matrix multiplication, Journal of the ACM 49(1), January 2002.
- [9] G. Myers, Approximately matching context-free languages, *Information Processing Letters*, 54, 1995.
- [10] S. Rajasekaran, Tree-adjoining language parsing in  $o(n^6)$  time, SIAM Journal on Computing, 25(4), 1996, 862-873.
- [11] S. Rajasekaran, S. Al Seesi, and R.A. Ammar, Improved algorithms for parsing ESLT-AGs: a grammatical model suitable for RNA pseudoknots, *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (TCBB), 7(4), 2010, pp. 619-627.
- [12] B. Saha, Efficiently computing edit distance to Dyck language, manuscript, November 2013.
- [13] L.G. Valiant, General context-free recognition in less than cubic time, *Journal of Computer and System Sciences* 10, 1975, 308-315.
- [14] D.H. Younger, Recognition and parsing of context-free languages in time  $n^3$ , Information and Control 10, 1967, 189-208.
- [15] U. Zwick, All pairs shortest paths in weighted directed graphs-exact and almost exact algorithms, Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on, IEEE, 1998.