On Repairing Structural Problems In Semi-structured Data

Flip Korn, Barna Saha, Divesh Srivastava
AT&T Labs–Research
{flip,barna,divesh}@research.att.com

Shanshan Ying
Nat'l Univ Singapore
shanshan@comp.nus.edu.sg

ABSTRACT

Semi-structured data such as XML are popular for data interchange and storage. However, many XML documents have improper nesting where open- and close-tags are unmatched. Since some semi-structured data (e.g., Latex) have a flexible grammar and since many XML documents lack an accompanying DTD or XSD, we focus on computing a syntactic repair via the edit distance.

To solve this problem, we propose a dynamic programming algorithm which takes cubic time. While this algorithm is not scalable, well-formed substrings of the data can be pruned to enable faster computation. Unfortunately, there are still cases where the dynamic program could be very expensive; hence, we give branch-and-bound algorithms based on various combinations of two heuristics, called MinCost and MaxBenefit, that trade off between accuracy and efficiency. Finally, we experimentally demonstrate the performance of these algorithms on real data.

1. INTRODUCTION

Semi-structured data provides a flexible representation where data can be nested as a tree and thus is very widely used, from XML documents to JSON data interchange files to annotated linguistic corpora. At the same time, this flexibility makes it prone to errors. A recent study of XML documents on the Web found that 14.6% of them (out of a 180K sample) were not well-formed, the majority of cases due to either open- and close-tag mismatches or missing tags [12]. The source of such errors is due to multiple factors including manual input [21], dynamically-generated data from faulty scripts [16], mapping and conversion errors (e.g., XML to relational mapping, MS Powerpoint 2007 converted to Powerpoint 2010); and interleaving of multiple sources (e.g., BGPmon pub-sub system which receives XML streams from multiple routers).

Often there is no known grammar associated with the data to test for validity; for example, only 25% of XML documents on the Web have an accompanying DTD or XSD [12]. Inferring one is a notoriously difficult problem [4], often requiring a whole repository rather than a single document, and which for some classes of documents is not even be possible [10]. Therefore, most existing work assumes that the document is well-formed and tests validity based on a supplied grammar [18, 17, 5].[1]

In this paper, we consider the problem of repairing an arbitrary semi-structured document into one that is well-formed. We believe this problem is in itself interesting for a variety of reasons. First, some existing documents, such as Latex, have a very flexible grammar that basically only requires proper nesting. Second, in the absence of a grammar, it may be "safer" to repair based on well-formedness rather than making domain-specific assumptions. Third, since well-formedness is a pre-condition for validity, well-formed repairs may serve as candidates for the user to choose from similar to the way word processors suggest autocorrection.

While *verifying* well-formedness in semi-structured data can be done straightforwardly, using a stack, in time linear in the size of the document, it is a much more challenging problem to *repair* a malformed document. Some existing tools, such as modern Web browsers, use simple rule-based heuristics to rectify mismatching tags. Perhaps the most common rule, employed by some web browsers such as Internet Explorer, is to substitute a matching close-tag whenever the current close-tag does not match the open-tag on the stack. However, a single extra or missing close-tag is enough to set off a cascade, requiring many close-tags to be replaced (or deleted). For example, `<A> <B> <C> <D> </E> </D> </C> </B> </A>` would require four substitutions, followed by the deletion of `</A>`. Another commonly used rule is to insert a matching close-tag whenever the current close-tag does not match, but this can trigger a similar cascade: the example above would require four insertions, followed by five deletions of the original close-tags.

Instead, we use the standard edit distance with insertion, deletion and substitution operators as a model for repair. The edit distance is used for modeling (and correcting) errors in many applications from information retrieval to computational biology. Note that `<A> <B> <C> <D> </E> </D> </C> </B> </A>` can be repaired in only one edit by deleting `</E>`. Furthermore, the text embedded within semi-structured documents sometimes follows certain patterns. For example, many XML documents only allow text to occur surrounded by matching open-close tags and require the existence of text between every adjacent matching pair. Here we consider how to exploit embedded text to aid in finding a more judicious repair via a constrained edit distance function.

Example: Figure 1(a) displays an example XML document of a bibliographic entry that is not well-formed: the `<authors>` open tag does not have a matching close tag; `<affiliation>` occurs out of place and is missing a matching tag; and the `</title>` close tag is out of order, occurring after `<authors>`. Figure 1(b) shows the document after the substitution rule-based heuristic is applied, requiring 3 substitutions and 2 insertions. A

[1]Exceptions to this are specifically tailored for HTML documents.

```
<article>
    <title>
      A Relational Model for Large Shared Data Banks
      <authors>
    </title>
      <author>
        <name> E. F. Codd </name>
        IBM <affiliation>
      </author>
</article>
```

(a) original document

```
<article>
    <title>
      A Relational Model for Large Shared Data Banks
      <authors>
      </authors>
      <author>
        <name> E. F. Codd </name>
        IBM <affiliation> </affiliation>
      </author>
    </title>
</article>
```

(b) rule-based repair

Figure 1: An Example and Rule-based Repair

well-formed repair based on tags only with fewest edits is given in Figure 2(a), which has edit distance 2: delete `<authors>` and delete `</affiliation>`. A well-formed repair based on tags and text with edit distance 3 is given in Figure 2(b): delete `<authors>`, insert `<affiliation>` before `IBM` and substitute `<affiliation>` for `</affiliation>`. Note that the latter repair consists of more edits than for tags only. □

In this paper, we introduce the problem of repairing malformed data based on two variants of well-formedness. Existing work has largely ignored this issue and focused on validating already well-formed documents according to some supplied grammar. Our solution is not just a first step towards validity but, in fact, interesting in its own right.

Our contributions are as follows. We give a dynamic programming algorithm which computes the (optimal) edit distance. Since this algorithm is cubic in the size of the input, regardless of the number of errors, it does not scale to large documents. Therefore, we present linear-time greedy algorithms that find close to optimal (within 10%) repairs in practice; these methods are significantly more accurate (more than an order of magnitude) than the rule-based heuristics. We also propose branch-and-bound algorithms for when multiple repairs are desired (such as for an autocorrection menu), since the dynamic program and greedy algorithms are geared towards finding a single repair. We present a variety of methods, with various trade-offs in accuracy and running time, whose performance depends on the number of edits rather than the length of the input. We perform a thorough experimental study to investigate these strategies on real data.

2. RELATED WORK

While we are not aware of prior work that specifically addresses the problem of repairing malformed data to make it syntactically well-formed, there is some work on repairing XML documents to make them valid with respect to a given DTD [5, 19, 20, 22]. However, these papers all assume the input is already well-formed. It is not clear how the techniques used in these papers, such as computing the tree or graph edit distance between a document and a DTD, can be applied to the problem here where documents are malformed. Some existing tools such as `Beautiful Soup`, `HTML Tidy` and `NekoHTML` allow for malformed HTML input and exploit domain knowledge to make them valid; however, they employ simple rules that can result in verbose repairs.

In the absence of a grammar, a natural approach would be to infer one for making repairs. For example, a few papers have been written about using a repository of XML documents to generate a DTD [9, 3] or XSD [4]. Unfortunately, grammar inference is a notoriously difficult problem, requiring a large number of examples and for which is impossible for some classes of grammars when only positive examples are given [10].

The problem of computing the edit distance from a string to a supplied context-free grammar has been studied; since the grammars for our notions of well-formedness can be expressed using a CFG, these existing solutions can be applied. Aho and Peterson [1] gave an $O(|G|^2n^3)$ algorithm which was later improved to $O(|G|n^3)$ by Myers [15], where n is the length of the input and $|G| = \sum_{A\to\alpha\in G}(|\alpha|+1)$ is the size of the grammar. Our dynamic program runs in $O(n^3)$ time, independent of grammar size.

It has been shown that *a non-deterministic version* of the language of well-formed bracketed strings is, in terms of parsing, the hardest CFG [11]. It is also known that parsing an arbitrary CFG is at least as hard as boolean matrix multiplication [13]. Therefore, computing the edit distance to a well-formed string in much less than cubic time would be a significant accomplishment.

Verifying well-formedness is a related but much easier problem: it is straightforward to do this using a stack in linear time. The problem is non-trivial, however, on streaming data where trading off accuracy (where distance to well-formedness is measured by Hamming distance) can allow this in sublinear space [14]. Other papers study the problem of validity checking: using a DTD or XML Schema, report if a given input document conforms to the given grammar. Some of these papers (e.g., [18]) perform *strong validation*, checking for well-formedness along with validity, while others (e.g., [17]) perform *weak validation*, assuming the input is already well-formed. In fact, our techniques can be used for preprocessing malformed input to enable application of the latter work.

Finally, we mention that our work fits into the context of data cleaning to satisfy database integrity constraints, including consistency under functional dependencies [2], inclusion dependencies [6] and record matching [8]. These papers all use edit distance as a notion of a minimal cost repair.

3. TAGS-ONLY CASE

We assume that the input data is tokenizable by a lexical analyzer and has been preprocessed into a sequence of *brackets*. For example, these brackets could correspond to the open-tags (<...>) and close-tags (</...>) of an XML document; to the curly braces or square brackets (and accompanying object name) of a JSON file; to a Latex file containing `\begin`{...} and `\end`{...}; etc.[2]

Definition 1. The *congruent* of a bracket x is defined as its symmetric opposite bracket, denoted $\bar{x}$. The congruent of a set of brackets X, denoted $\bar{X}$, is defined as $\{\bar{x} \mid x \in X\}$.

We assume a bracket namespace is not given *a priori*. Let R and S denote the sets of brackets obtained from the two directions

[2]We shall ignore all other components in the XML document besides tags for now such as attributes and text and treat empty-element tags as two tags, e.g., `<a/>` becomes `<a></a>`.

```
<article>
     <title>
       A Relational Model for Large Shared Data Banks
     </title>
     <author>
       <name> E. F. Codd </name>
       IBM
     </author>
</article>
```

(a) considering tags only

```
<article>
     <title>
       A Relational Model for Large Shared Data Banks
     </title>
     <author>
       <name> E. F. Codd </name>
       <affiliation> IBM </affiliation>
     </author>
</article>
```

(b) considering tags and text

Figure 2: Two Possible Repairs

(i.e., open and close) after tokenization, respectively. We shall use $T = R \cup \bar{S}$ to denote the set of *open brackets* and $\bar{T} = \bar{R} \cup S$ the *close brackets*. (Each $x \in T$ has exactly one congruent $\bar{x} \in \bar{T}$ and vice versa.)

Definition 2. A *match* between two brackets x and y, denoted $x \asymp y$, occurs when $x \in T$, $y \in \bar{T}$ and $y = \bar{x}$.

Consider a string $s = s_1 ... s_n$, of length n, over some bracket alphabet T, that is, $s \in (T \cup \bar{T})^*$.

Definition 3. A *well-formed string* over some bracket alphabet T obeys the context-free grammar G_T with productions $S \rightarrow SS$, $S \rightarrow \varepsilon$ and $S \rightarrow aS\bar{a}$ for all $a \in T$.[3] So $|G_T| = 4|T| + 3$.

Example: Let $T = \{a, b, c\}$. Then $ab\bar{b}c\bar{c}\bar{a}$ is a well-formed string, since it can be parsed as $S \rightarrow aS\bar{a} \rightarrow a(SS)\bar{a} \rightarrow a(bS\bar{b})(cS\bar{c})\bar{a} \rightarrow ab(\varepsilon)\bar{b}c(\varepsilon)\bar{c}\bar{a} \rightarrow ab\bar{b}c\bar{c}\bar{a}$. However, $ab\bar{a}\bar{b}$ is not well-formed. ☐

Definition 4. A *well-formed bracketed language* $L(G_T)$ over some bracket alphabet T is the set of strings from $T \cup \bar{T}^*$ accepted by the grammar G_T defined above.

Definition 5. The *edit distance* $E(s, s')$ between two strings s and s' is the minimum number of insertions, deletions and substitutions needed to transform s into s', where an insertion of a after position i transforms $s_1 ... s_i s_{i+1} ... s_n$ to $s_1 ... s_i a s_{i+1} ... s_n$; a deletion at position i transforms $s_1 ... s_{i-1} s_i s_{i+1} ... s_n$ to $s_1 ... s_{i-1} s_{i+1} ... s_n$; and a substitution to a at position i transforms $s_1 ... s_{i-1} s_i s_{i+1} ... s_n$ to $s_1 ... s_{i-1} a s_{i+1} ... s_n$.

Definition 6. The *Bracketed Language Edit Distance Problem*, given string s, is to find $\arg\min_{s'} E(s, s')$ such that $s' \in L(G_T)$.

We shall henceforth use the term *edit distance* of a string to mean the edit distance from the string to a well-formed repair.
Example: The edit distance of $ab\bar{a}\bar{b}$ to a well-formed string is 2. For example, it can be changed to $a\bar{a}$ using 2 deletions and $ab\bar{b}\bar{a}$ using 2 substitutions. ☐

3.1 Optimal Solution

Algorithm 1 presents pseudocode for a dynamic program to compute the edit distance $\min_{s'} E(s, s')$ to a well-formed string s'. The algorithm runs in $O(n^3)$ time and requires $O(n^2)$ space, where n is the string length and is based on applying the different grammar productions to the input. Given some substring $s_i ... s_j$, $j > i$, either s_i and s_j could be edited to matching brackets (the $S \rightarrow aS\bar{a}$ production) or else the string could be broken into two adjacent well-formed substrings (the $S \rightarrow SS$ production). Let $B[i, j]$ be the cost of editing s_i and s_j to match. When $j = i$, $B[i, j] = B[i, i] = 1$, also $C[i, j] = C[i, i] = 1$. Hence, the recurrence is

$$C[i,j] \leftarrow \min(B[i,j]+C[i+1,j-1], \min_{i \leq k \leq j-1} C[i,k]+C[k+1,j])$$

The following claim establishes the correctness of the recursion and hence, also of Algorithm 1.[4]

CLAIM 1. *Algorithm 1 correctly finds the edit distance, given string s, such that it is accepted by G_T.*

PROOF. Consider any substring of length 1, s_i. The minimum edit distance is 1, which can be achieved either by deleting s_i (by rule $S \rightarrow \varepsilon$), or by inserting matching open/close bracket. For substrings of length 2, $s_i s_{i+1}$, $B[i, i+1]$ is the computed edit distance and $C[i, i+1] = B[i, i+1]$. These serve as the base cases.

Suppose, by induction hypothesis, Algorithm 1 correctly computes minimum edit distance for all substrings of length at most l. We now take any substring of length $l+1$. W.l.o.g, let it be $s_1 s_2 .... s_{l+1}$. Consider s_1. An optimal algorithm has the following options.

1. Deletes s_1 and the minimum edit distance is $1 + C[2, l+1]$. This option is considered in Algorithm 1, where for $k = i$, $i = 1$, the algorithm has the choice to set edit distance as $C[1, l+1] = C[1, 1] + C[2, l+1]$, and both the terms on RHS are computed correctly by the algorithm according to base case and induction hypothesis.

2. Matches s_1 to some s_j, $j > 1$, possibly by substitution. There are again few subcases here.

   (a) $j = l+1$ and $s_1 \asymp s_{l+1}$. In that case, the minimum edit distance is $C[1, l+1] = C[2, l]$. The algorithm in this case correctly computes $B[1, l+1] = 0$ and has the choice to set edit distance as $C[1, l+1] = B[1, l+1] + C[2, l]$; by induction hypothesis, the second term in RHS is computed correctly.

   (b) $j = l+1$, $s_1 \in T$ and substituted to match s_{l+1}. In that case, the minimum edit distance is $C[1, l+1] = 1 + C[2, l]$. The algorithm in this case correctly computes $B[1, l+1] = 1$ and has the choice to set edit distance as $C[1, l+1] = B[1, l+1] + C[2, l]$; by induction hypothesis, the second term in RHS is computed correctly.

   (c) $j = l+1$, $s_{l+1} \in \bar{T}$ and substituted to match s_1. In that case, the minimum edit distance is $C[1, l+1] =$

[3] Some instances of well-formedness additionally require that the document is nested within a single open-close pair, i.e., $s \in T(T \cup \bar{T})^* \bar{T}$, but we dispense with this for simplicity.

[4] Full version with proofs at `http://www.cs.umd.edu/~barna/vldb13-xmlrepair.pdf`

$1 + C[2, l]$. The algorithm in this case correctly computes $B[1, l + 1] = 1$ and has the choice to set edit distance as $C[1, l + 1] = B[1, l + 1] + C[2, l]$; by induction hypothesis, the second term in RHS is computed correctly.

(d) $j = l + 1$, $s_1 \in \bar{T}$ and $s_{l+1} \in T$, both deleted or substituted to match each other. In that case, the minimum edit distance is $C[1, l + 1] = 2 + C[2, l]$. The algorithm in this case correctly computes $B[1, l + 1] = 2$ and has the choice to set edit distance as $C[1, l + 1] = B[1, l + 1] + C[2, l]$; by induction hypothesis, the second term in RHS is computed correctly.

(e) $j < l + 1$. In that case, the minimum edit distance is $C[1, l + 1] = C[1, j] + C[j + 1, l + 1]$. This option is considered by the algorithm where both the terms in RHS are correctly computed by induction hypothesis.

The above options are exhaustive, Algorithm 1 considers all the options, computes edit distance correctly in all these cases and returns the minimum. Hence, Algorithm 1 computes minimum edit distance for any substring of length $l + 1$ correctly, therefore, by induction, the proof is established. □

For ease of exposition, we do not show how to construct an s'. A single minimum cost repair can be constructed from the dynamic programming tableau straightforwardly. However, constructing multiple repairs having minimum cost is non-trivial. We defer this discussion until Section 3.2.

Algorithm 1 Dynamic Program for Edit Distance

```
Require: tokenized string $s = s_1 ... s_n$
Ensure: edit distance $\min_{s'} E(s, s')$ where $s'$ is well-formed
 1: for all $\ell$ from 1 to $n - 1$ do
 2:     for all $i$ from 1 to $n - \ell$ do
 3:         $j \leftarrow i + \ell$
 4:         $C[i, j] \leftarrow B[i, j] + C[i + 1, j - 1]$
 5:         for all $k$ from $i$ to $j - 1$ do
 6:             $C[i, j] \leftarrow \min(C[i, j], C[i, k] + C[k + 1, j])$
 7:         end for
 8:     end for
 9: end for
10: return $C[1, n]$
```

The cubic growth in running time as a function of string size becomes a problem for large strings. Luckily, one can often prune away well-formed substrings to speed up the running time. Using a stack, this can be done straightforwardly in linear time by recursively finding matching adjacent pairs and removing them. We note, however, that this may eliminate some candidate repairs from consideration. For example, given the string $a\bar{a}\bar{b}\bar{a}$, the only optimal repair with pruning that can be found is $a\bar{a}a\bar{a}$ whereas $ab\bar{b}\bar{a}$ also has an edit distance of 1. Nonetheless, there exists at least one repair of the pruned string with edit distance equal to that optimal for the original string.

CLAIM 2. *Well-formed substrings removal preserves the edit distance. That is, the edit distance of the pruned string is the same as the edit distance for the full string.*

PROOF. We do an induction on the number of edits. For strings with edit distance 0, well-formed substring pruning returns the empty string with edit distance 0. This serves as the basis. Considering the claim to be true for all strings with minimum edit distance less than d, we next take any string s with minimal edit distance d. Consider an optimal algorithm that defers doing the first edit as much as possible without affecting the optimality, and let the d edit positions be $p[1] < p[2] < ... < p[d]$. Also, let t be the string that results from s after removing the well-formed substrings. Let the prefix in t corresponding to $s_1 s_2 ... s_{p[1]}$ be $t_1 t_2 ... t_q$. Clearly, $t_1, ..., t_{q-1}$ correspond only to open brackets.

Case 1: If $s_{p[1]}$ is not part of any well-formed substring then t_q corresponds to $s_{p[1]}$. Consider performing the same edit operation as the optimal algorithm at $s_{p[1]}$ and also at t_q. The resultant string s' after the edit at $s_{p[1]}$ in s, has edit distance $d - 1$. If well-formed substrings are removed from s', and also from t after the edit at t_q to get t', then they both return the same processed string. By the induction hypothesis, s' and t' have the same edit distance. Edit distance of s and t are one more than edit distance of s' and t' respectively, hence, both s and t have the same edit distance.

Case 2: Otherwise $s_{p[1]}$ is part of a well-formed substring w. There are few cases to consider based on whether $s_{p[1]} \in T$ or $s_{p[1]} \in \bar{T}$. The main idea is to show that in all these cases, there exists an alternate edit script such that $s_{p[1]}$ is not part of any well-formed substrings.

Subcase 2a: First, consider $s_{p[1]} \in T$. If we remove well-formed substring from the prefix $s_1 s_2 ... s_{p[1]}$, then we must have $t_1 t_2 ... t_q s_l s_{l+1} ... s_{p[1]}$ for some integer $l \geq 1$, where all $t_1, t_2, ..., t_q, s_l, s_{l+1}, ..., s_{p[1]}$ are open brackets. The edit at $s_{p[1]}$ can be either delete, or substitution to some different open bracket, or substitution to closed bracket. For the ith symbol in w, let $match[i]$ denote the position of the symbol in s to which it matches in w. If edit at $s_{p[1]}$ is delete, then $s_{match[p[1]]}$ must be matched with some s_{r_1}, $r_1 < p_1$, otherwise edit costs at $s_{p[1]}$ and $s_{match[p[1]]}$ is 2, both of which can be saved resulting in lower edit distance and contradicting the optimality. If s_{r_1} is part of $t_1 t_2 ... t_q$, then we can simply instead match $s_{p[1]}$ with $s_{match[p[1]]}$ and delete s_{r_1}. This reduces the edit distance by 1 and we can use the same argument as in Case 1. Else s_{r_1} is part of $s_{l+1} ... s_{p[1]-1}$. Then $s_{match[r_1]}$ must be matched with some s_{r_2}. Again, if s_{r_2} is part of $t_1 t_2 ... t_q$, then we can instead delete s_{r_2} and match $s_{p[1]}$ with $s_{match[i]}$ and s_{r_1} with $s_{match[r_1]}$. Otherwise, s_{r_2} is part of $s_{l+1} ... s_{p[1]-1}$ and we continue in this fashion until we reach some s_{r_h} in $t_1 t_2 ... t_q$. We then instead can delete $s_{r[h]}$ and match the corresponding pairs in w for $s_{l+1}, ..., s_{p[1]}$.

Now, if edit at $s_{p[1]}$ is a substitution to some different open bracket to match with, say, $s_{r'_1}$, $r'_1 \neq match[p[1]]$, then the substring $s_{p[1]+1} ... s_{r'_1 - 1}$ is not well-formed and needs at least one more edit. In stead, if $s_{r'_1}$ is not part of w, we could have just deleted $s_{r'_1}$ and matched all of w to save one in edit distance. Else, if $s_{r'_1}$ is part of w then by matching w instead, we could have lower the edit distance by 2–this contradicts the optimality. Hence, no optimal algorithm considers substituting at $s_{p[1]}$ a different open bracket. Similar argument removes the possibility of substituting a closed bracket at $s_{p[1]}$.

Subcase 2b: Now consider $s_{p[1]} \in \bar{T}$. If we remove well-formed substring from the prefix $s_1 s_2 ... s_{p[1]}$, then we must have $t_1 t_2 ... t_q s_l s_{l+1} ... s_{l'} s_{p[1]}$ where $s_{l'} \asymp s_{p[1]}$. If edit at $s_{p[1]}$ is delete then $s_{l'}$ must be matched with some s_{r_1}, $r_1 > p[1]$. Instead, it is possible to match $s_{l'}$ and $s_{p[1]}$, and delete s_{r_1}. This defers the first edit without affecting the optimality giving a contradiction. Edit at $s_{p[1]}$ to a substitution to a different closed bracket is also not possible because that requires editing $s_{l'}$ as well. If edit at $s_{p[1]}$ is a substitution to a new open to match say $s_{r'_1}$, $r'_1 > p[1]$. Then again we can delete $s_{r'_1}$ and match $s_{l'}$ and $s_{p[1]}$, contradicting the fact that the considered optimal algorithm defers the first edit as much as possible. □

There are instances where well-formed substring pruning will not be very effective; for example, consider the string $abcded\bar{d}\bar{c}\bar{b}\bar{a}$. Here edit distance is 1, but since there is no well-formed substring, nothing can be eliminated. We investigate its effectiveness on real data in Section 5.

3.2 Incremental Approach

The dynamic program presented in the previous section has two deficiencies. The first is that it has the same running time regardless of how many errors exist in the input string; that is, its best-case running time is as slow as the worst-case. Second, it can extract a single edit script associated with the edit distance found but does not provide a natural way of enumerating multiple repairs. Here we describe branch-and-bound strategies, with various trade-offs between accuracy and running time, that are affected only by the number of errors, not the length of the string, and are capable of incrementally reporting repairs. Our algorithms are based on various combinations of greedy heuristics. All of our methods maintain a stack that, at any point, contains open brackets remaining to be matched with close brackets from the string.

As a warm-up, we consider the case where $|T| = 1$. Here it turns out we can apply recursive matching of adjacent pairs to obtain a sequence of zero or more elements from $\bar{T}$ followed by a sequence of zero or more elements from T, that is, $\bar{a}^*a^*$. Then the minimum cost repair, for the close bracket and open bracket substrings separately, is obtained by applying substitutions to make adjacent pairs match and, if a singleton remains, delete it. So if the pruned string is $\bar{a}^i a^j$, the resulting edit distance is $\lceil i/2 \rceil + \lceil j/2 \rceil$. Hence, for $|T| = 1$ the edit distance can be computed in $\Theta(n)$ time. Clearly the same edit distance can be obtained via many different matchings, precisely $p(i/2) \times p(j/2)$ when i and j are even, where $p(k)$ is the partition function denoting the number of ways to write k as a sum of positive integers. Rather than enumerating all these possibilities, a single canonical form such as only the minimum- or maximum-depth nesting shall be reported.

When $|T| \geq 2$, repairs of these two scenarios are similar to the $|T| = 1$ case: adjacent pairs are examined and the appropriate substitutions are made to make these adjacent pairs match. There is an additional type of error than can occur besides these two: empty stack with remaining close brackets and a non-empty stack when the string has terminated. There could be an open bracket of one type followed by a close bracket of another type.

It is this third scenario that is most challenging. Note that each possible insertion operation has a symmetrically equivalent deletion operation, so for the sake of reducing the enumerated repairs we shall use a canonical form involving only deletions. Given a mismatch of types between an adjacent open-close pair, there are only five edit operations that need to be considered:

1. Delete the open bracket on the left;
2. Delete the close bracket on the right;
3. Substitute the left or right bracket to make a matching pair;
4. Substitute the open bracket on the left to a close bracket;
5. Substitute the close bracket on the right to an open bracket.

For the third alternative, we shall canonically replace the right close bracket to match the left open bracket. For the last two alternatives, the way the replacement bracket is chosen is as follows. For an open bracket substituted to a close bracket, we assign the bracket matching the next open bracket on the stack. (We only consider this alternative when the stack remains non-empty after deleting the open bracket.) For a close bracket substituted to an open bracket, we wait to assign the bracket until the first close bracket is encountered that gets paired with it (until then it is a "ghost" open) and then assign it so that the pair matches; if the string terminates before such a pairing occurs then it gets resolved to a deletion rather than a substitution.

The following claim establishes the correctness of our algorithm.

CLAIM 3. *By considering edit operations at only one of the following scenarios, a sequence of choices exists that leads to the optimal edit distance: (1) an empty stack when a close bracket occurs; (2) a non-empty stack when the string has terminated; and (3) an open bracket of one type adjacent to a close bracket of a different type. Furthermore, exhaustive branching to the five edit alternatives above leads to an optimal repair.*

PROOF. From Claim 2, we know, removal of well-formed substring preserves the edit distance. If the original string has edit distance 0, then removal of well-formed substring results in an empty string, in all other cases, the resultant string is non-empty. Suppose, the original string has edit distance d and is denoted by s^d. The corresponding string after removal of well-formed substrings is denoted by s'^d. We know s'^d is non-empty and does not contain any well-formed substrings in it. Then, it must be the case, that either (1) $s_1'^d \in \bar{T}$, or (2) there is a sequence of open brackets in s'^d followed by string termination, or (3) a sequence of open brackets in s'^d followed by a closed bracket which does not match the current open bracket at the stack top. Any optimal algorithm must make an edit operation in all these scenarios. In case (1), the only possible edits are (i) deletion of the closed bracket and (ii) substitution of the closed bracket to an open bracket. In case (2), any canonical repair by pairing up the open brackets and deleting at most one open bracket gives the optimal solution. In case (3), there are five possible edit operations (i) deleting the open bracket on the left, (ii) deleting the close bracket on the right, (iii) substituting the left or right bracket to make a matching pair, (iv) substituting the open bracket at the top of the stack to a close bracket to match the next open bracket on the stack and (v) substituting the current close bracket on the string to some open bracket. Our exhaustive branching algorithm considers all these options. If an optimal algorithm does any of the above edit operations except (v) in case (3), and results in string s^{d-1} with edit distance $d-1$, then there is a branch that considers the exact same repair followed by well-formed substring removal, resulting in s'^{d-1}. Now, we can do an induction to establish the claim. Now suppose, the optimal algorithm considers option (v) in case (3) and does a substitution to some $x \in T$. In our branching algorithm, we have a branch that substitutes the current close bracket to a *ghost* open bracket. This ghost open bracket can be resolved to "any" open bracket at that point in the algorithm, and hence to x as well. Therefore, the resultant string after well-formed substring removal has edit distance $d-1$ and we can apply induction. The type of ghost open is resolved if and only if at some point removal of well-formed substrings requires it to be matched with a given closed bracket. Since, removal of well-formed substring preserves the edit distance, this type-resolution of the ghost open bracket, also does not increase the edit distance. If, we reach a state where the stack is non-empty with unresolved ghost opens when the string terminates, then it indicates that the particular branch is not optimal. In order to get a valid repair, it is enough to resolve the ghost opens to deletion. □

We consider two heuristics for choosing from these alternatives, the first of which makes a greedy decision based on local information and the second of which is based on non-local information but ignores interleaving between bracket types:

- **MaxBenefit:** At each mismatch, consider all five alternatives and take the one that enables the largest well-formed substring to be pruned (the size of which is the benefit). The time to test these alternatives is amortized: an alternative resulting in a larger number of matched brackets takes longer time but also advances that much further along, requiring in total linear space and time. When $|T| = 1$, MaxBenefit finds an optimal cost repair.

- **MinCost:** Precompute the imbalance for each bracket type subsequence (similar to the $|T| = 1$ case) as follows. Let $\sigma_a(s)$, for $a \in T$, denote the subsequence of s containing brackets a or $\bar{a}$. For each $a \in T$ and each suffix of $\sigma_a(s)$, we find the remaining subsequence after matching pairs elimination. Suppose the result for $\sigma_a(s)$ is $\bar{a}^i a^j$ and that there are currently k open brackets a on the stack. Then the number of unbalanced brackets in $\sigma_a(s)$ is $|k - i| + j$. Taking all the subsequences $\sigma_a(s)$, for each $a \in T$, the minimum number of edit operations to well-formedness (via substitutions) is $\lceil (\sum_{a \in T} |k_a - i_a|)/2 \rceil + \lceil (\sum_{a \in T} j_a)/2 \rceil$. This gives a lower bound on the edit distance. So at each mismatch, the alternatives are considered in turn and the one which best improves the lower bound is chosen. This strategy can be done in linear space and time since the imbalance counts (for each subsequence suffix) can be precomputed and stored globally. When $|T| = 1$, MinCost also finds an optimal cost repair.

Unfortunately, both of these heuristics can result in approximations of the edit distance with a performance ratio that is linear in n. For example, the string $aaaaaabbbb\bar{a}\bar{a}\bar{a}\bar{a}\bar{a}\bar{a}$ could result in 8 edit operations with MaxBenefit if the wrong alternative among ties is chosen at each mismatch (at mismatch of type $b\bar{a}$, substitute $\bar{a}$ to a, instead of substituting b to $\bar{b}$), and the string $abcde\bar{a}\bar{e}\bar{d}\bar{c}\bar{b}$ could result in 8 edit operations with MinCost if the wrong alternative among ties is chosen at each mismatch. To mitigate this, all of the ties can be maintained in a queue and tried as part of a branch-and-bound algorithm; we call these variants MaxBenefit ++ and MinCost ++, respectively.

Interestingly, MinCost performs well on the hard input for MaxBenefit and MaxBenefit performs well on the hard input for MinCost: the first string can be repaired in 2 operations using MaxBenefit and the second in 2 operations using MinCost. Therefore, we consider hybrid strategies which combine MaxBenefit and MinCost in various ways to complement each other. In particular, we consider the following three hybrids.

- **Conservative:** Try all the choices in the union that Max-Benefit or Min-Cost gives.

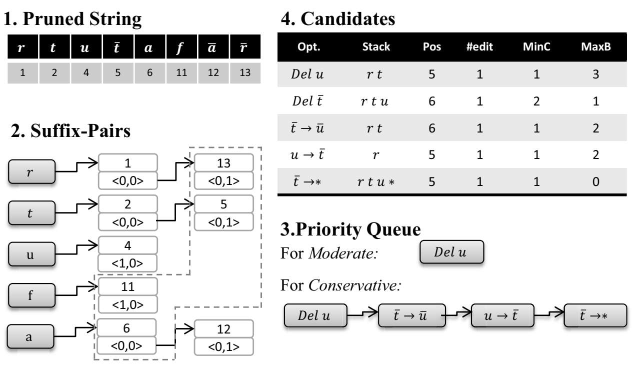
- **Moderate:** Try all the choices in the multiset intersection of the choices that Max-Benefit or Min-Cost gives; if the intersection is empty, then try all the choices from their union.

- **Liberal:** Select one choice at random from the multiset intersection of the choices that Max-Benefit or Min-Cost gives; if the intersection is empty, then select one choice at random from their union.

We shall investigate the trade-offs between accuracy and running time of these strategies in Section 5.

3.3 Implementation for Branching Strategies

Figure 3 illustrates how the branch-and-bound algorithm works on the tokenized string $rtu\bar{t}an\bar{n}f\bar{a}\bar{r}$ from Figure 1(a). Two global structures, bracket list and suffix pairs, are preprocessed in a single



1. Pruned String

r	t	u	$\bar{t}$	a	f	$\bar{a}$	$\bar{r}$
1	2	4	5	6	11	12	13

4. Candidates

Opt.	Stack	Pos	#edit	MinC	MaxB
$Del\ u$	$r\ t$	5	1	1	3
$Del\ \bar{t}$	$r\ t\ u$	6	1	2	1
$\bar{t} \rightarrow \bar{u}$	$r\ t$	6	1	1	2
$u \rightarrow \bar{t}$	r	5	1	1	2
$\bar{t} \rightarrow *$	$r\ t\ u\ *$	5	1	1	0

Figure 3: Illustration of Branch-and-Bound Methods

scan to assist the procedure. The bracket list contains the tokenized brackets and their index positions after well-formed substring pruning. The suffix pairs are constructed based on the suffix of the string starting from position pos, and for the subsequence $\sigma(x)$ corresponding each type of bracket x, where a pair $\langle i, j \rangle$ represents the remaining $\bar{x}^i x^j$ after well-formed substring pruning. For example, bracket r has two suffix pairs: $\langle 0, 0 \rangle$ at position 1 and $\langle 0, 1 \rangle$ at 13.

The first mismatch occurs at $\bar{t}$ (position 5). For each of the five edit alternatives, we list the stack state, the string position after the edit is applied and the total number of edits incurred for the repaired prefix. In addition, we show the MaxBenefit and MinCost values for each alternative. Taking option *Del u* for instance, t and $\bar{t}$ also get matched, so the total benefit value is 3. To get the MinCost value, we consider the suffix pairs surrounded by dashed lines. With r and t in the stack, *Del u* reduces the imbalance from the suffix pair $\langle 1, 0 \rangle$ at position 4 for u by 1; the pairs $\langle 0, 1 \rangle$ at 13 for r and $\langle 0, 1 \rangle$ at 5 for t get canceled out by the stack; and of the remaining pairs, $\langle 0, 0 \rangle$ at 6 for a gives 0 (since brackets can potentially match) and $\langle 1, 0 \rangle$ at 11 for f gives 1, resulting in a total value of 1.

We show the resulting priority queues for Moderate and Conservative at the bottom right of Figure 3. The former contains the intersection of alternatives where MinCost of 1 is the lowest and MaxBenefit of 3 is the highest; the latter contains the union of alternatives having MinCost of 1 or MaxBenefit of 3. Candidates are inserted into the priority queue sorted on ascending order of (#edit+MinCost); this provides a lower bound on the eventual repair cost. By visiting nodes in this order, it is guaranteed that any fully repaired string must have edit distance no larger than the existing partial repairs will have after completion.

Following the Moderate priority queue, the next mismatch occurs at $\bar{a}$, for which the alternative chosen is *Del f*, with MinCost value of 0 and MaxBenefit value of 5. The final output is $rt\bar{t}an\bar{n}\bar{a}\bar{r}$ with cost of 2. Conservative returns the same repaired string with the same edit distance but uses more space and time for the extra candidates generated.

To speed up the algorithm, we can avoid visiting candidates with the same stack state and string position but having larger cost. This can done by hashing the candidate's stack and the (index of the) remaining string suffix; when multiple repairs are needed, the hashing function is based on the repaired prefix rather than the stack.

4. TAGS WITH TEXT CASE

A commonly occurring pattern for semi-structured documents, especially those used for data interchange, is that text content must occur between all matching leaf-level tags and only between such tags, that is, it cannot occur immediately before or after any intermediate (non-leaf) tag. This is the norm for JSON files as well as XML files encoding JSON. Moreover, a study of DTDs on the Web revealed that only 1% of XML data exchange documents allowed so-called *mixed content elements* (allowing both text and tags) [7]. Indeed, we can exploit this pattern to compute a more judicious repair.

Let Σ be some alphabet and $W = \{w \mid w \in \Sigma^+\}$ denote a set of words that can be embedded in a semi-structured document. We assume that the input data has been preprocessed into a sequence of brackets and words (which assumes the existence of markers that tell the lexical analyzer how to distinguish between brackets and words).

Definition 7. A *well-formed string* over some bracket alphabet T with embedded text from W obeys the context-free grammar $G_{T,W}$ with productions $S' \to S \mid \varepsilon$ and $S \to SS \mid xS\bar{x} \mid xw^+\bar{x}$, for all $x \in T$ and where all $w \in W$.

Definition 8. A *well-formed bracketed language with text* $L(G_{T,W})$ over some set of words W and bracket alphabet T is the set of strings from $(W \cup T \cup \bar{T})^*$ accepted by the grammar $G_{T,W}$ defined above.

Example: Let $W = \{w\}$ and $T = \{a, b\}$. Then $abw\bar{b}\bar{a}$ is a well-bracketed string, since it can be parsed as $S \to aS\bar{a} \to abw\bar{b}\bar{a}$ but $ab\bar{b}w\bar{a}$ is not. ☐

Definition 9. The *Bracketed Language with Text Edit Distance Problem*, given a string $s \in (W \cup T \cup \bar{T})^*$, is to find $\arg\min_{s'} E(s, s')$ such that $s' \in L(G_{T,W})$. Here we allow insertion, deletion and substitution operations on brackets but do not allow any operations on words, that is, the words in a string must remain as they are.

Example: The string $ab\bar{b}w\bar{a}$ has edit distance 2 and can be repaired to $abw\bar{b}\bar{a}$ using one deletion and one insertion. ☐

4.1 Optimal Solution

As shown in Figure 2, if we consider only the bracket subsequence of the string and apply the tags-only methods from Section 3 to find a solution, the resulting repairs may not obey $G_{T,W}$. In fact, $G_{T,W}$ is strictly more constrained than G_T (hence, the edit distance for tags-only lower-bounds that for tags with text). Therefore, we need to design a new algorithm.

Algorithm 2 presents the pseudocode for the dynamic program. As with the tags-only case, the algorithm runs in $O(n^3)$ time and requires $O(n^2)$ space. Given some substring $s_i...s_j$, the algorithm first checks if it contains a word (that is, $s_k \in W$ for some $k \in [i, j]$) and, if not, deletes $s_i...s_j$, resulting in cost $C[i, j] \leftarrow j - i + 1$. Otherwise, either s_i and s_j could be edited to matching brackets surrounding a well-formed substring (the $S \to xS\bar{x}$ production); or s_i and s_j could be edited to matching brackets surrounding a sequence of one or more words, after deleting all brackets in the substring $s_{i+1}..s_{j-1}$, denoted by $D[i + 1, j - 1]$ (the $S \to xw^+\bar{x}$ production); or else the string could be broken into two adjacent well-formed substrings (the $S \to SS$ production).

Let $B[i, j]$ be the smallest cost of editing s_i and s_j to match. When $j = i$, if $s_i \in W$, $C[i, j] = C[i, i] = 2$, else $C[i, j] = C[i, i] = 1$. For all other cases, while the substring $s_i s_{i+1}...s_j$ contains a word, the recurrence is

$$C[i,j] \leftarrow \min \begin{cases} B[i,j] + C[i+1, j-1], \\ B[i,j] + D[i+1, j-1], \\ \min_{i \le k \le j-1} C[i,k] + C[k+1, j] \end{cases}$$

If $s_i \in W$ and $s_j \in \bar{T}$, then $B[i, j] = 1$. Similarly, if $s_j \in W$ and $s_i \in T$, then $B[i, j] = 1$. Else if, $s_i \in W$ and $s_j \in T$, then $B[i, j] = 2$. Also, if $s_j \in W$ and $s_i \in \bar{T}$, or both s_i and s_j are words, then $B[i, j] = 2$.

Algorithm 2 Dynamic Program for Tags with Text

```
Require: tokenized string $s = s_1...s_n$
Ensure: edit distance $\min_{s'} E(s, s')$ where $s'$ is well-formed
 1: for all $\ell$ from 1 to $n - 1$ do
 2:   for all $i$ from 1 to $n - \ell$ do
 3:     $j \leftarrow i + \ell$
 4:     if $s_k \notin W, \forall k \in [i, j]$ then
 5:       $C[i, j] \leftarrow j - i + 1$
 6:     else
 7:       $C[i, j] \leftarrow B[i, j] + D[i + 1, j - 1]$
 8:       $C[i, j] \leftarrow \min(C[i, j], B[i, j] + C[i + 1, j - 1])$
 9:       for all $k$ from $i$ to $j - 1$ do
10:         $C[i, j] \leftarrow \min(C[i, j], C[i, k] + C[k + 1, j])$
11:       end for
12:     end if
13:   end for
14: end for
15: return $C[1, n]$
```

CLAIM 4. *Algorithm 2 correctly finds the edit distance, given a string s, such that it is accepted by $G_{T,W}$.*

PROOF. Consider any substring of length 1, s_i. If $s_i \in W$, since words cannot be deleted, two matching open and close parenthesis need to be inserted to surround s_i, resulting in edit distance 2. If $s_i \in T$, the minimum edit distance is 1, which can be achieved by deleting s_i (by rule $S \to \varepsilon$), or by inserting matching open/close bracket. For substrings of length 2, $s_i s_{i+1}$, $B[i, i + 1]$ is the computed edit distance and $C[i, i+1] = B[i, i+1]$. These serve as the base cases. Note that, if $s_i \in W$ and $s_{i+1} \in barT$ or, $s_{i+1} \in W$ and $s_i \in T$, then $B[i, i+1] = 1$. In all other cases, $B[i, i+1] = 2$.

Suppose, by induction hypothesis, Algorithm 2 correctly computes minimum edit distance for all substrings of length at most l. We now take any substring of length $l + 1$. W.l.o.g, let it be $s_1 s_2....s_{l+1}$. If, there is no word in $s_1 s_2...s_{l+1}$, then edit distance is $l + 1$. Otherwise, consider s_1. An optimal algorithm has the following options.

1. Deletes s_1 if s_1 is not a word and the minimum edit distance is $1 + C[2, l + 1]$. This option is considered in Algorithm 2, where for $k = i$, $i = 1$, the algorithm has the choice to set edit distance as $C[1, l + 1] = C[1, 1] + C[2, l + 1]$, and both the terms on RHS are computed correctly by the algorithm according to base case and induction hypothesis.

2. Matches s_1 to some s_j, $j > 1$, possibly by substitution. There are again few subcases here. First, consider when neither s_1 nor s_{l+1} is a word.

   (a) $j = l+1$ and $s_1 \asymp s_{l+1}$. In that case, the minimum edit distance is $C[1, l + 1] = \min C[2, l], D[2, l]$. The algorithm in this case correctly computes $B[1, l+1] = 0$ and has the choice to set edit distance as $C[1, l + 1] = \min B[1, l + 1] + C[2, l], B[1, l + 1] + D[2, l]$; by induction hypothesis, the second term in RHS is computed correctly.

   (b) $j = l + 1$, $s_1 \in T$ and s_{l+1} is substituted to match s_1. In that case, the minimum edit distance is

$C[1, l+1] = \min 1 + C[2, l], 1 + D[2, l]$. The algorithm in this case correctly computes $B[1, l+1] = 1$ and has the choice to set edit distance as $C[1, l+1] = \min B[1, l+1] + C[2, l], B[1, l+1] + D[2, l]$; by induction hypothesis, the second term in RHS is computed correctly.

(c) $j = l + 1$, $s_{l+1} \in \bar{T}$ and s_1 is substituted to match s_{l+1}. In that case, again the minimum edit distance is $C[1, l+1] = \min 1 + C[2, l], 1 + D[2, l]$. The algorithm in this case correctly computes $B[1, l+1] = 1$ and has the choice to set edit distance as $C[1, l+1] = \min B[1, l+1] + C[2, l], B[1, l+1] + D[2, l]$; by induction hypothesis, the second term in RHS is computed correctly.

(d) $j = l + 1$, $s_1 \in \bar{T}$ and $s_{l+1} \in T$, both deleted or substituted to match each other. In that case, the minimum edit distance is $C[1, l+1] = \min 2 + C[2, l], 2 + D[2, l]$. The algorithm in this case correctly computes $B[1, l+1] = 2$ and has the choice to set edit distance as $C[1, l+1] = \min B[1, l+1] + C[2, l], B[1, l+1] + D[2, l]$; by induction hypothesis, the second term in RHS is computed correctly.

(e) $j < l + 1$. In that case, the minimum edit distance is $C[1, l+1] = C[1, j] + C[j+1, l+1]$. This option is considered by the algorithm where both the terms in RHS are correctly computed by induction hypothesis.

If one or both of s_i and s_{l+1} are words, then again algorithm correctly computes $B[i, l+1]$. If $s_i \in W$ and $s_{l+1} \in \bar{T}$, then $B[i, l+1] = 1$ can be achieved by inserting a matching open bracket before s_i. Similarly, if $s_j \in W$ and $s_i \in T$, then $B[i, j] = 1$ can be achieved by inserting a matching close bracket after s_{l+1}. Else if, $s_i \in W$ and $s_{l+1} \in T$, then $B[i, j] = 2$ by substituting s_{l+1} to some close bracket and inserting matching open bracket in front of s_i. Also, if $s_j \in W$ and $s_i \in \bar{T}$, then $B[i, l+1] = 2$ by substituting s_i to some open bracket and inserting matching close bracket after s_{l+1}. If both s_i and s_{l+1} are words, then $B[i, j] = 2$ can be obtained by inserting matching open and close brackets before and after s_i and s_{l+1} respectively. Again Algorithm 2 considers all the possible options as in the case for both s_i and s_{l+1} belonging to tags, and correctly computes the edit distance in each case. These options are exhaustive, Algorithm 2 considers all the options, computes edit distance correctly in all these cases and returns the minimum. Hence, Algorithm 2 computes minimum edit distance for any substring of length $l + 1$ correctly, therefore, by induction, the proof is established. □

Similar to the tags-only case, we can preprocess the string to enable faster computation by removing well-formed substrings using a stack. We must also mark the presence of each removed substring to allow local edit operations, such as surrounding matching brackets that would not be allowed otherwise. This pruning can speed up the dynamic program by shortening the input string. Unfortunately, it does not guarantee a repair with optimal edit distance. For example, the string $aaaw\bar{a}\bar{a}aw\bar{a}\bar{a}\bar{a}\bar{a}$ has edit distance 1 (by replacing the second $\bar{a}$ to a) but the pruned string $aw\bar{a}\bar{a}\bar{a}$ has edit distance 2. We show that the above dynamic program stays within at least a factor of 2 after well-formed substring pruning.

CLAIM 5. *Removing well-formed substrings obtains a 2-approximation on the edit distance.*

PROOF. Define by a block a substring of brackets that appears between two consecutive occurrence of texts, or before the first text, or after the last text. Consider a new edit distance function, which is exactly similar to the original edit distance problem, except that two consecutive open brackets (close brackets) in a block can both be deleted at a cost of 1. If this new edit distance function has minimum distance d, then clearly, the optimal edit distance for the original problem cannot be more than $2d$ (we pay 2 to delete two open (close) brackets in a block). The main insight is to consider this new function, and show that well-formed substring pruning preserves edit distance for it, and hence guarantees 2-approximation to the original problem.

Let s be the original string and s' be the string obtained after removing well-formed substrings from s. Let decompose s and write it as concatenation of substrings $s_1 r_1 s_2 r_2 ... s_k r_k$, where $s' = s_1 s_2 ... s_k$ and each of $r_i, i = 1, 2, .., k$ are well-formed substrings, with the possibility of r_k being empty. In s, consider within each block, the substrings of the form $T^* \bar{T}^*$, which are well-formed according to the tag-only grammar. Since, these substrings are not well-formed according to text+tag grammar, they are part of s' as well. Consider an optimal algorithm for text+tag that works on s. Perform the same edits done by this algorithm on these substrings both on s and s', and let the resultant strings be t and t'. Clearly, it is enough to show for our proof that edit distance of t and new edit distance of t' are the same. Next consider the texts in t that are not surrounded by matching open and close brackets and consider the edits done by the optimal algorithm to make them surrounded by at least one matching open and close bracket. Clearly, we can perform the same edits on t' as well to make sure each text is surrounded by at least one open and close bracket. If the resultant strings after these edits are z and z', then it is enough to show that edit distance of z and new edit distance of z' are identical.

Each block, except the first and the last blocks of z and z' have the structure of $\bar{T}^+ T^+$. The first block has the structure of T^+, whereas the last block has the structure of $\bar{T}^+$. Now consider only the tags in z (z') and consider an optimal algorithm for tag-only case, which always prefers deletion over substitution whenever possible. Such an algorithm will never make substitutions within the same block of z (z') to match a $\bar{T}$ with a T– the algorithm can simply delete both of these tags at the same cost of 2. Therefore, the only substitutions that the algorithm can possibly do within a single block must happen solely within $\bar{T}^+$ or T^+. This substitution cost is the same cost as deletion for the new edit distance where two consecutive open or close tags can be deleted at a cost of 1. All the other edits are either inter-block or consists of intra-block deletes. Therefore, the new edit cost of z (z') is at most the tag-only edit distance of z (z'). By, similar arguments, the tag-only edit distance of z (z') is at most the new edit distance of z (z'). Hence the tag-only edit distance and new edit distance of z (z') are the same. Also, the well-formed substrings of z, considering only tags and considering both text and tags are identical. We know from Claim 2, that well-formed substring removal preserves tag-only edit distance–therefore, the new edit distance of z' is same as z and the proof is established. □

4.2 Incremental Approach

Once again, the dynamic program is intended for computing the edit distance, along with perhaps a single repair, but does not provide a natural way of enumerating multiple repairs. Here we identify the different scenarios in which repairs should be handled as well as a set of possible repairs for each of these scenarios, similar to our approach for the tags-only case. We start by giving push-down automata in Figure 4 that allows a well-formed string to be verified, for both the tags-only and tags with text cases. Note that the only scenarios not handled in Figure 4(a) are precisely the three scenarios from Section 3: (1) a close bracket is read when the stack is empty; (2) a terminated string when the stack is non-empty; and (3) a close bracket in the string does not match the open bracket on the stack. We have shown that only considering these violations, rather than eagerly repairing the string, leads to the optimal edit distance while making the running time proportional to the number of errors rather than the size of the string. We apply the same idea to tags with text: only scenarios not covered by the automaton in Figure 4(b) are considered. We show how to deal with these scenarios in Table 1.

State	*Token*	*Action*	*Next State*
0	open	push to stack and advance string	1
0	close	sub close in string to ghost open	0
		delete close from string	0
0	word	insert ghost open before word	0
0	null		4
1	open	push to stack and advance string	1
1	close	sub open from stack to close and pop matching pair from stack, if possible	1
		pop open from stack, if possible	1
		sub close from string to ghost open	1
		delete close from string	1
1	word		2
1	null	clean-up	4
2	open	insert matching close to string before open	2
		sub open in string to matching close	2
		delete open from string	2
2	match	pop from stack and advance string	3
2	close	sub string to match stack	2
		insert matching close in string, if possible	2
		if next token is word, delete string close	2
		else push matching open to stack	2
2	word		2
2	null	insert matching close to string	2
3	open	push to stack and advance string	1
3	match	pop from stack and advance string	3
3	close	sub close in string to match stack	3
		delete close in string	3
		pop open from stack	3
		sub open in stack to close and pop resulting match from stack, if possible	3
		sub close in string to ghost open	3
3	word	insert ghost open in string (before word)	3
3	null	clean-up if non-empty stack	4

Table 1: State Transition Table for Tags with Text Case

By contrast, there are nine scenarios based on the automaton for tags with text in Figure 4(b). Furthermore, the alternatives for open-close bracket mismatch depend on which state the mismatch occurs in. If this occurs in State 3 then we consider the following alternatives (similar to tags-only): make the open and close brackets match via substitution; pop the open bracket from the stack; delete the close bracket in the string; substitute the open bracket in the stack to close; or substitute the close bracket in the string to ghost-open (equivalently, substitute the right to match the left). However, if this occurs in State 2, then there are different options since the word(s) must be surrounded by a pair of brackets: make the open and close brackets match via substitution; insert a matching close bracket, if possible; or delete the close bracket if the next token in the string is a word (otherwise insert a matching open bracket).

Any input string can be partitioned into *blocks* of brackets separated by text. There are five additional scenarios in addition to the three for tags-only: (4) a close bracket occurs immediately after an open bracket; (5) an open bracket occurs immediately after a word; (6) a word occurs immediately after a close bracket; (7) a word occurs as the first token; and (8) the string terminates in a word. For these additional scenarios, there are various edit alternatives, which are listed in Table 1.

The so-called "clean-up" phase referred to in States 1 and 3 of Table 1 is invoked if the stack is non-empty when the string terminates. In this case, the goal is to take the existing stack, paying attention to the blocks that each stack open bracket is part of, and perform the minimum number of substitutions and deletions to obtain a well-formed string. For example, suppose there are three blocks on the stack, the first with ab, the second with cde and the third with fg. By deleting d and replacing c with $\bar{b}$, f with $\bar{e}$ and g with $\bar{a}$, the resulting brackets are well-formed. We omit details for lack of space.

Example: Let $s = a\bar{a}\bar{a}waaaw\bar{a}\bar{a}\bar{a}$. Repair scenario (4) occurs after the pair $a\bar{a}$ in the first block (since there is no text separating them), causing $\bar{a}$ to either be replaced with an open bracket or deleted (the other two alternatives from State 1 are not possible). Scenario (4) occurs again at the next $\bar{a}$ in the string with the same edit alternatives. Suppose we choose the substitution alternative both times. Then Scenario (6) occurs after the first w, which we can repair by either inserting a close, substituting the open to a close or deleting the open. Suppose we choose to insert a close. The remaining elements will be read without problem until the string terminates, at which point the stack will be non-empty with two open brackets. At that point, they must both be deleted since they both occurred in the same block. The final repair, then, is $aaaw\bar{a}aaaw\bar{a}\bar{a}\bar{a}$ with a cost of 5. Had we instead chosen the alternative to delete the close brackets in Scenario (4), the string would have been repaired to $aw\bar{a}aaaw\bar{a}\bar{a}\bar{a}$ at a cost of 3, which is optimal. □

The following claim establishes the correctness of the automaton.

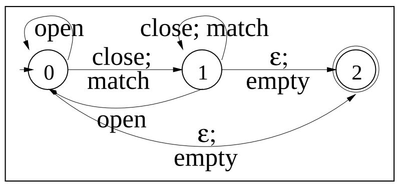
CLAIM 6. *The automaton given in Table 1 with branches to all the above edit alternatives, obtains a 2-approximation on edit distance.*

PROOF. The proof is similar to Claim 3, therefore, we provide a sketch here. If we remove well-formed substrings greedily, then at any point being stuck in the algorithm implies either we have i) an empty stack with close bracket on the string (error at State 0), or ii) only open brackets followed by string termination or followed by close brackets within a single block (error at State 1), or iii) mismatched close brackets with open brackets on stack (error at State 2). The algorithm takes all possible edit options at each of the error state, and therefore, there must be one branching that is optimal for the new edit function defined in Claim 5. This branch reduces the new edit distance by 1 and hence by induction, there exists a branch leading to optimal cost for the new function. Since, any optimal algorithm for the new edit function returns a solution within twice the minimum edit distance of our problem, the claim is established. □

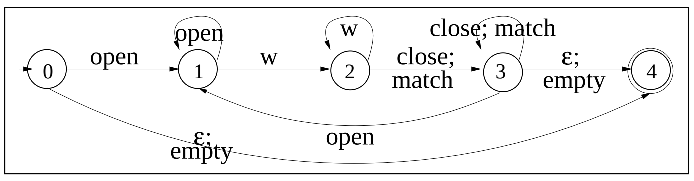
As before, for the MaxBenefit strategy, all alternatives are considered in turn and then the one resulting in the largest number of brackets that can be paired to matches is chosen. For MinCost, we employ the same cost estimation formula as the tags-only case, since it provides a lower bound; the alternatives are sorted with respect to these costs.

5. EXPERIMENTAL EVALUATION

This section gives a thorough experimental evaluation of methods for both Tags-Only and Tags With Text problems, against the *Dynamic Programming* (DP) and *branch-and-bound* algorithms.



(a) tags-only



(b) tags with text

Figure 4: Automata for tags-only and tags with text

5.1 Set-up

All algorithms were implemented in Java and executed on a server with a Quad-Core AMD Opteron(tm) Processor 8356@1 GHz and 128 GB RAM running Centos 5.8.

We used the following two real data sets:

- *BGP*[5] real-time routing information provided by BGPmon. We used a portion of the stream output over some time interval.
- *Tree Bank*[6] annotated linguistic text, with average depth 7.8 and max depth 36. We extracted random substrees with maxdepth no less than 20 and merged them together.

Both data sets normally satisfy the grammar $G_{T,W}$. For the tags-only methods, only the tag subsequences are retained and for tags with text the subsequence of tags and text was retained; all attributes were removed.

Error Model: We injected errors into the data selected uniformly at random from the operations listed in Table 2.

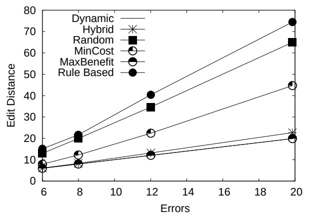
Table 2: Types of Errors

Operation	Description
Delete(i)	delete the tag at the i-th position
Insert(i,*a*)	insert tag *a* to the i-th position, where a is randomly chosen from $(T \cup \bar{T})$
Swap(i, j)	swap the tag located at the i-th position with the one at the j-th
Flip(i)	change the i-th tag to close (resp., open) if it is open (resp., close)
Sub(i,*a*)	substitute the i-th tag with a, where a is randomly selected from $(T \cup \bar{T})$
DeepInsert(*a*,*h*)	insert tag a into some position i having $depth(i) > h$, where a is randomly selected from $(T \cup \bar{T})$

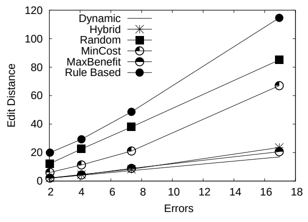
Metrics: Each experiment was repeated 100 times, with each string containing a different set of errors; we report *Average Running Time* and *Average Edit Distance*.

5.2 Tags-Only, Single Repairs

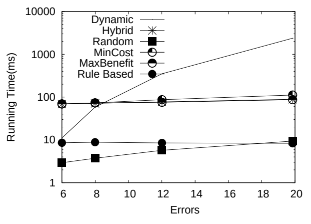
We compared methods that are designed to return a single repair, including DP as well as MaxBenefit, MinCost and Hybrid, where Hybrid randomly picks one choice either given by MinCost or by Maxbenefit. As a baseline, we also tried three rule-based heuristics for handling open-close mismatches: one which performs a substitution to make them match; one which deletes the open; and one which deletes the close (the best of which on our data was the substitution rule, so we use that in experiments). Finally, we also tried five trials of Random which randomly chooses one from the five alternatives reporting the lowest edit distance among these. By default, well-formed substring pruning is applied to speed up the methods.



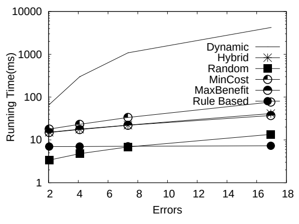
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



(d) On *Tree Bank* data

Figure 5: Single Repair, *Error Number* (*TagsOnly*)

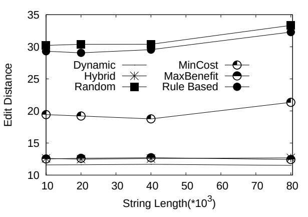
Figure 5 shows the edit distance and running time as a function of the number of errors. For *BGP*, the errors ranged from 6 to20 with initial string length fixed at $40,000$; after well-formed substring pruning, the string length was significantly reduced to the range $[20, 160]$. For *Tree Bank*, the error ranged from 2 to16 with string length fixed at $4,000$; after pruning the length was reduced to $[40, 280]$.

While DP gave the smallest edit distance (it is optimal), it was also the slowest in almost all cases. The running time of DP increased significantly with the number of errors, even with well-formed substring pruning: the string length after pruning increased from 20 to 160 on *BGP* and from 40 to 280 on *Tree Bank*. Rule-Based, in contrast, ran the fastest but gave the highest edit distance. Interestingly, Rule-Based was even less accurate than Random (recall that Random chooses from among five alternatives while Rule-Based makes a single deterministic choice), which in turn also affected its running time performance due to the additional edits. The edit cost of MaxBenefit was close to optimal, its inaccuracy growing with increasing errors but at a very slow rate; at the same time, its running time was 1-2 orders of magnitude faster than that of DP. MinCost, on the other hand, was no faster but much less accurate than MaxBenefit. Hybrid, which integrates both heuristics, is slightly less accurate than MaxBenefit but slightly faster.

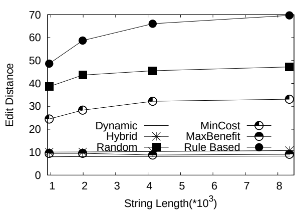
Figure 6 demonstrates the scalability with respect to string length, with the number of errors fixed at around 12 on *BGP* and 8 on *Tree Bank*. Due to well-formed substring pruning, the average

[5]http://bgpmon.netsec.colostate.edu/

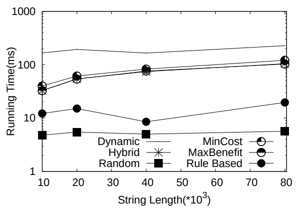
[6]http://www.cis.upenn.edu/˜treebank/



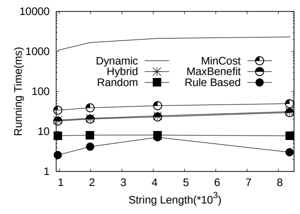
(a) On *BGP* data



(b) On *Tree Bank* data



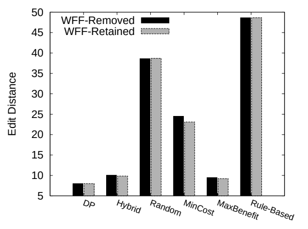
(c) On *BGP* data



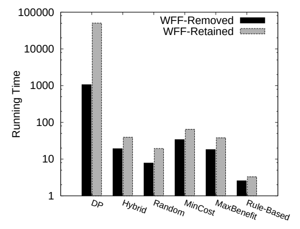
(d) On *Tree Bank* data

Figure 6: Single-Repair, *String Length* (*TagsOnly*)

string length was reduced to around 60 on *BGP* and 150 on *Tree Bank* from all the initial string lengths. Hybrid and MaxBenefit follow DP closely in accuracy and outperform the latter in running time, as much as 1-2 orders of magnitude.



(a) Edit Distance



(b) Running Time

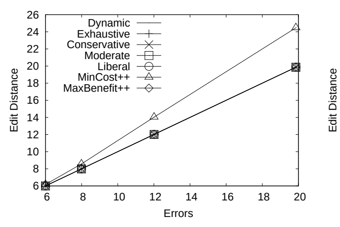
Figure 7: Well-formed Substring (*TagsOnly*)

As shown in Sec. 3.1, well-formed substring pruning, does not affect the edit distance of DP. However, it does affect its running time. We ran a set of experiments on *Tree Bank* with average string length of $1,000$ and with 8 errors on average. The bars in Figure 7 show the difference in running time and edit distance with and without such pre-processing. The edit distance for DP and Rule-Based stays the same, but there is a slight difference in other methods, which can be explained by randomization. For DP the running difference is quite noticeable (from $1066ms$ to $50510ms$) while, for others, the increase in running time is small; such increase is largely brought by the cost in building the global suffix stack.

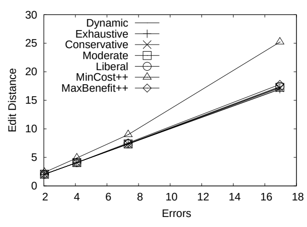
5.3 Tags-Only, Multiple Repairs

We compared various branch-and-bound methods: Exhaustive (tries all five choices at each branch point), Conservative, Moderate, Liberal, MinCost++ and MaxBenefit++. The key difference between these methods is the number of alternatives tried at each branch, where there is an inherent trade-off between accuracy and running time.

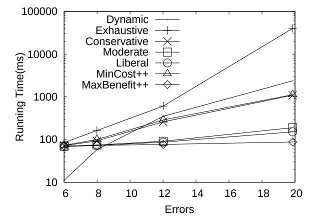
Single-Repair Performance We begin by showing the results for single repair, i.e.,$K = 1$, using DP as a baseline. Figure 8 shows performance versus error number ranging from 6 to 20 on *BGP* and 2 to 16 on *Tree Bank*. As expected, Exhaustive gives optimal edit distance while MinCost++ is the least accurate one. In



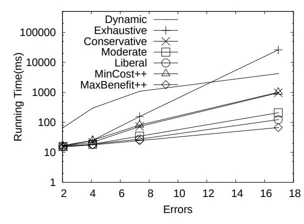
(a) On *BGP* data



(b) On *Tree Bank* data

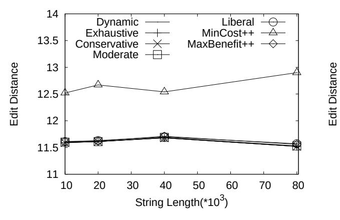


(c) On *BGP* data

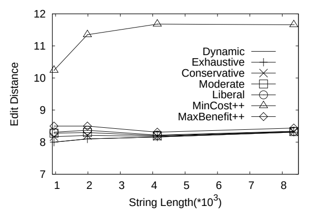


(d) On *Tree Bank* data

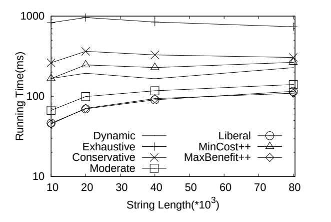
Figure 8: Multi-Repairs, *Error Number* (*TagsOnly*)



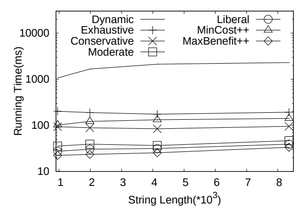
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



(d) On *Tree Bank* data

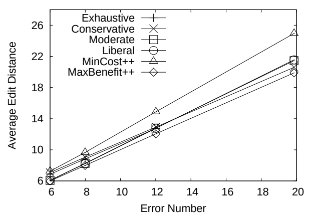
Figure 9: Multi-Repair, *String Length* (*TagsOnly*)

Figure 8(a) to (d), all methods except MinCost++ and Exhaustive are almost as accurate as DP but much faster. MaxBenefit++ beats all other methods in running time, but is less accurate.

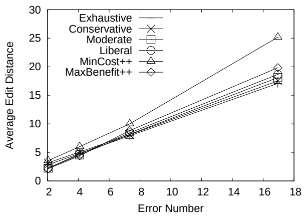
Figure 9 shows performance versus string length ranging from $10,000$ to $80,000$ (with roughly 12 errors) on *BGP*, and $1,000$ to $8,000$ (with roughly 8 errors) on *Tree Bank*. After pre-processing, the string sizes were greatly reduced to 60 on *BGP* and 130 on *Tree Bank*. When a string is short with many errors, DP wins; otherwise, Exhaustive is faster. In general, the branch-and-bound methods were not greatly affected by string length and perform well when the number of errors is small.

Multi-Repairs Performance Figure 10 illustrates the performance for finding 5 repairs when the number of errors increases from 6 to 20 on *BGP* (with string length fixed at 40,000) and from 2 to 16 on *Tree Bank* (with string length fixed at 4,000). When error number is 20 on *BGP* dataset, it takes Exhaustive an extremely large amount of time, so we do not plot the results there.

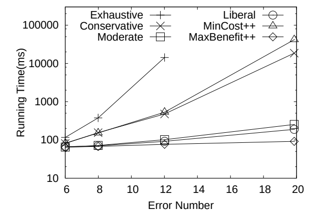
Figure 11 shows running time when string length grows from 10,000 to 80,000 on *BGP* and 1,000 to 8,000 on *Tree Bank*. With



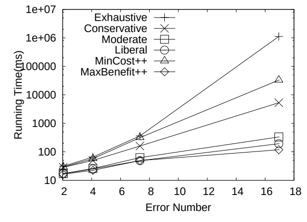
(a) On *BGP* data



(b) On *Tree Bank* data

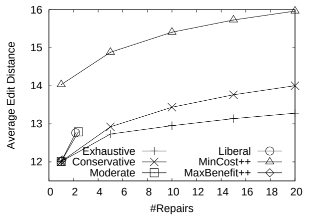


(c) On *BGP* data

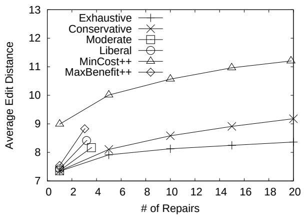


(d) On *Tree Bank* data

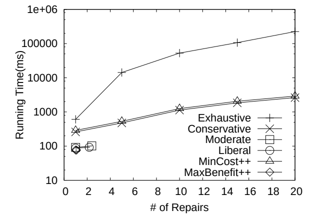
Figure 10: *top-5* Repairs, *Error Number* (*TagsOnly*)



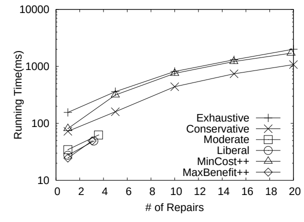
(a) On *BGP* data



(b) On *Tree Bank* data

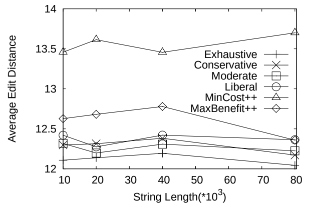


(c) On *BGP* data

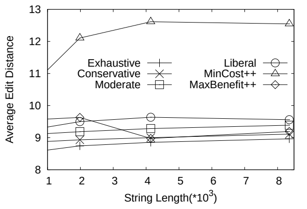


(d) On *Tree Bank* data

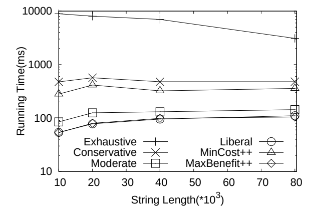
Figure 12: *top-k*, Scalability (*TagsOnly*)



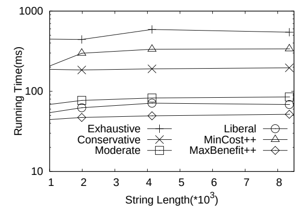
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



(d) On *Tree Bank* data

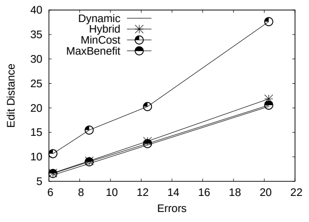
Figure 11: *top-5* Repairs, *String Length* (*TagsOnly*)

the well-formed substring removed, the string length decreases significantly to 130 on *Tree Bank*, and 60 on *BGP* dataset.

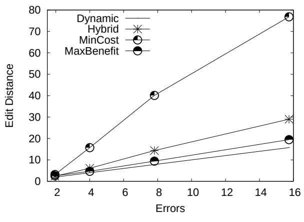
Exhaustive, Conservative and MinCost++ are 10 times slower than Moderate, Liberal and MaxBenefit++ since the former methods return more repairs than the latter do. MinCost++ is less accurate than Exhaustive and Conservative, but is faster on *BGP*; on *Tree Bank*MinCost++ is less accurate and comparable to Exhaustive in running time.

K-Repairs Performance We evaluated the performance of finding up to K repairs, for $K \in \{1, 5, 10, 15, 20\}$, with string length 40,000 for *BGP* and 4,000 for *Tree Bank*. Note that not all methods were able to obtain K repairs. Moderate, Liberal and MaxBenefit++ run faster by aggressively pruning; therefore, they result in fewer total repairs. Figure 12 shows that these three methods were unable to return more than 5 repairs. Only Exhaustive, Conservative and MinCost++ obtained up to 20 repairs. With more nodes visited, Exhaustive returned repairs lower in average edit distance but requires running time. On *Tree Bank*, MinCost++ is worse in both average edit distance and running time, which means **MinCost** prunes off some nodes low in edit distance, leading to a longer edit path and larger search space. For methods where there are enough repairs, the running time grows linearly in K.

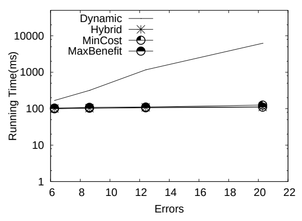
5.4 Tags with Text, Single Repair



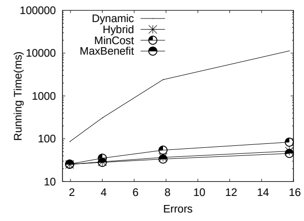
(a) On *BGP* data



(b) On *Tree Bank* data



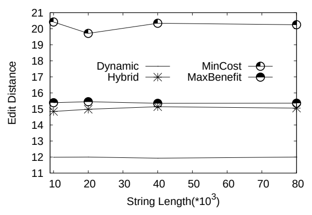
(c) On *BGP* data



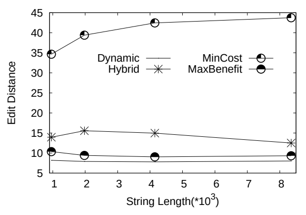
(d) On *Tree Bank* data

Figure 13: Single-Repair, *Error Number* (*TagsWithText*)

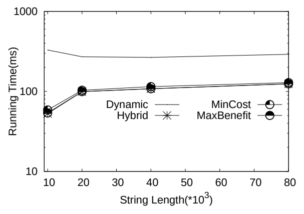
We ran experiments using the dynamic program for repairing documents with text (satisfying $G_{T,W}$ rather than G_T) as well as the following analogues of tags-only methods: MaxBenefit, MinCost and Hybrid. Figure 13 presents performance as a function of number of errors, ranging from 6 to 20 (with fixed length 40,000) on *BGP* and 2 to 16 (with fixed string length 4,000) on *Tree Bank*. The overall trend is similar to that of tags-only presented in Sec. 5.2. DP again is the slowest while MinCost is the least accurate. MaxBenefit is again both more accurate and faster than MinCost and Hybrid is slightly faster but less accurate than MaxBenefit. With the well-formed substring removed, the average string size in Figure 13 decreases from 40,000 to 100 $\sim$ 400 on *BGP*, and from 4,000 to 60 $\sim$ 500 on *Tree Bank*. The running time for DP grows quickly, due to the increase in string length, while other



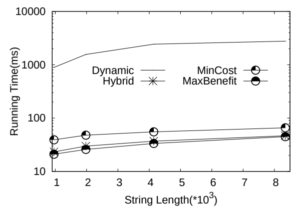
(a) On *BGP* data



(b) On *Tree Bank* data



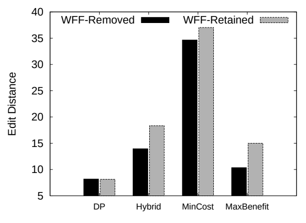
(c) On *BGP* data



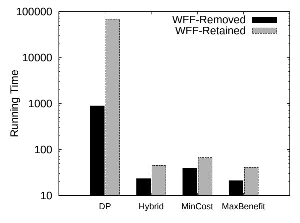
(d) On *Tree Bank* data

Figure 14: Single-Repair, *String Length* (*TagsWithText*)

methods are less affected by errors. MaxBenefit approximates DP well in edit distance and is faster by up to two orders of magnitude.



(a) Edit Distance



(b) Running Time

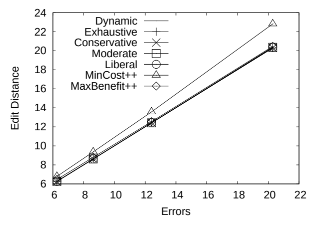
Figure 15: Well-formed Substring (*TagsWithText*)

Figure 15 exhibits the relative performance with and without well-formed substring pruning. There is an insignifcant decrease in edit distance for DP, from 8.19 to 8.14, but a significant increase in running time, from $1,000ms$ to $70,000ms$. MaxBenefit, MinCost and Hybrid exhibited a small increase in both edit distance and running time. (Recall that these methods are randomized, which may be partly responsible for part of the difference.)

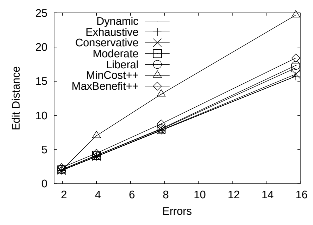
5.5 Tags with Text, Multiple Repairs

Single-Repair Performance Figure 16 gives performance versus number of errors, ranging from 6 to 20 on *BGP* (string length 40,000) and from 2 to 16 on *Tree Bank* (string length 4,000). The edit costs were close to optimal for all methods except MinCost++, especially on *BGP*. MaxBenefit++ was the fastest and DP was the slowest with one exception: when error number is 20 on *BGP* (where the string length is very small after pruning). On *Tree Bank*, when the error number is 16 and the string length is 500, Exhaustive still beats DP by an order of magnitude. In general, the advantages of branch-and-bound methods are seen with strings of large sizes and few errors.

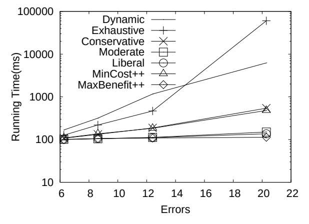
Figure 17 illustrates the scalability versus string length with roughly 12 errors for *BGP* and 8 errors for *Tree Bank*. Not surprisingly, Conservative was the second best after Exhaustive. MaxBenefit++ was superior to MinCost++ on *Tree Bank* but not on *BGP*, which shows that the heuristics they're based on are complementary. After pruning, the string length remains around 150 for *BGP*



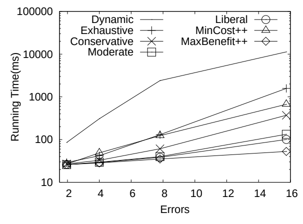
(a) On *BGP* data



(b) On *Tree Bank* data

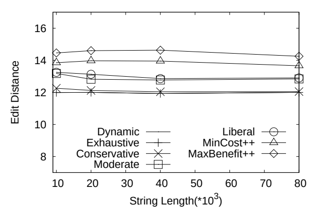


(c) On *BGP* data

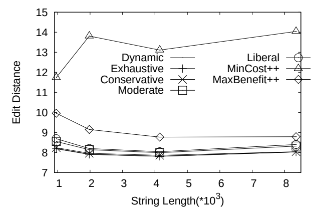


(d) On *Tree Bank* data

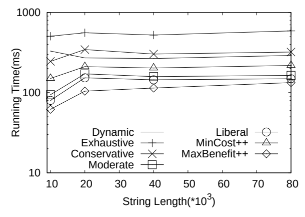
Figure 16: Multi-Repair, *Error Number* (*TagsWithText*)



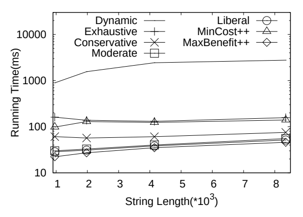
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



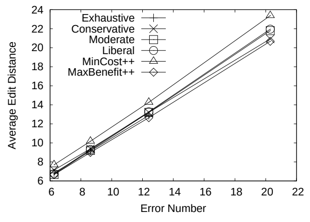
(d) On *Tree Bank* data

Figure 17: Multi-Repair, *String Length* (*TagsWithText*)

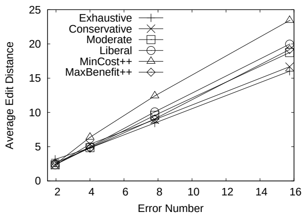
and 300 for *Tree Bank*, which explains the stability in running time for DP on both datasets. Nonetheless, DP is slower than Exhaustive by two orders of magnitude on *Tree Bank*.

Multi-Repair Performance Figures 18 and 19 give the performance and scalability of branching methods for finding 5 repairs. Again, MinCost++ consistently had the worst accuracy. On *Tree Bank*, Conservative beats MinCost++ on both accuracy and speed, which shows the MinCost heuristic does not work well in some cases as few of its branches led to low-cost repairs. The constancy in string length after pruning is the main reason why the running time for both datasets are fairly constant with increasing string size. The average edit distance of Moderate, Liberal and MaxBenefit++ seem smaller than even Exhaustive when string length equals to 8,000; however, this is partly due to them finding no more than 3 repairs.

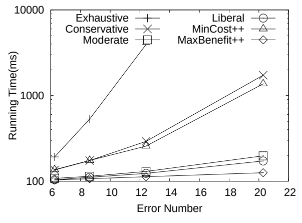
K-Repair Performance We issued queries to find K repairs for K between 1 to 20; the results are displayed in Figure 20. The methods that prune more aggressively failed to return as many as K repairs in some instances. Only MinCost++, Conservative and Ex-



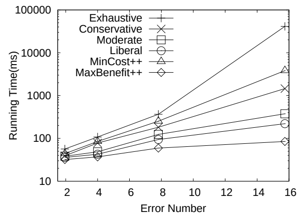
(a) On *BGP* data



(b) On *Tree Bank* data

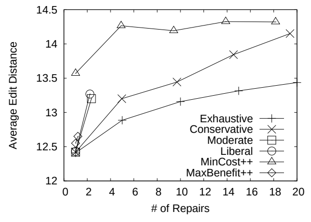


(c) On *BGP* data

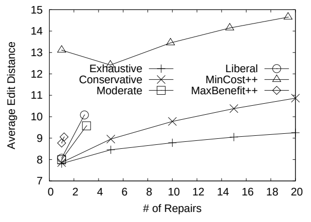


(d) On *Tree Bank* data

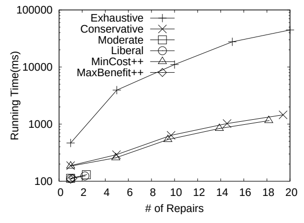
Figure 18: *top-5* Repairs, *Error Number* (*TagsWithText*)



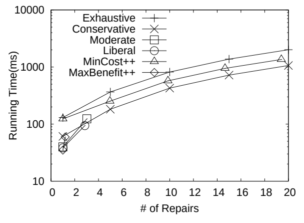
(a) On *BGP* data



(b) On *Tree Bank* data

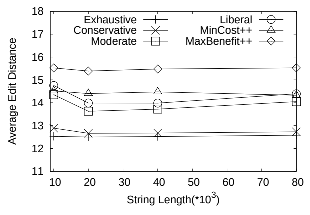


(c) On *BGP* data

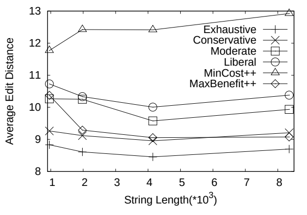


(d) On *Tree Bank* data

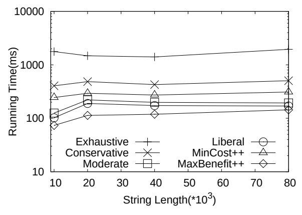
Figure 20: *top-k*, Scalability (*TagsWithText*)



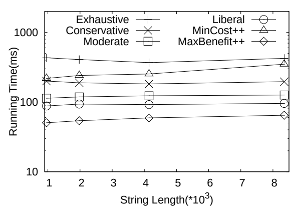
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



(d) On *Tree Bank* data

Figure 19: *top-5* Repairs, *String Length* (*TagsWithText*)

haustive were capable of returning K repairs. Exhaustive gave the smallest average edit distance but at a much higher running time; Conservative was comparable to MinCost++ but retrieved repairs with smaller average edit distance, especially on *Tree Bank*. With increasing K, the average edit distance increased slowly while the running time increased linearly, which indicates some degree of scability.

6. CONCLUSIONS

Future work includes considering more complicated edit distances, including approximate string matching for tag labels as well as more general sequence alignment operations and allowing additional operations such as swaps and moves. In addition, there are other cues in a document, even when the grammar is unknown, that can be leveraged for a more judicious repair including attributes, text correlations, etc.

7. REFERENCES

[1] A. V. Aho, T. G. Peterson: A Minimum Distance Error-Correcting Parser for Context-Free Languages. SIAM J. Comput. 1(4): 305-312 (1972)

[2] G. Beskales, I. F. Ilyas and L. Golab, Sampling the Repairs of Functional Dependency Violations under Hard Constraints. *PVLDB 3(1)*: 197-207 (2010)

[3] G. J. Bex, F. Neven, T. Schwentick, K. Tuyls, Inference of Concise DTDs from XML Data, *VLDB 2006*: 115-126

[4] G. J. Bex, F. Neven, S. Vansummeren, Inferring XML Schema Definitions from XML Data, *VLDB 2007*: 998-1009

[5] U. Boobna and M. de Rougemont: Correctors for XML Data. XSym 2004: 97-111

[6] P. Bohannon, M. Flaster, W. Fan and R. Rastogi, A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification. *SIGMOD 2005*: 143-154

[7] B. Choi, What are real DTDs like? *WebDB 2002*: 43-48

[8] W. Fan, J. Li, S. Ma, N. Tang and W. Yu, Interaction between record matching and data repairing. *SIGMOD Conference 2011*: 469-480

[9] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri and K. Shim: DTD Inference from XML Documents: The XTRACT Approach. IEEE Data Eng. Bull. 26(3): 19-25 (2003)

[10] E. M. Gold, Language Identification in the Limit, *Information and Control* 10(5): 447-474 (1967)

[11] S. A. Greibach, The Hardest Context-Free Language, *SIAM J. Comput.* 2(4): 304-310 (1973)

[12] S. Grijzenhout and M. Marx, The quality of the XML web. *CIKM 2011*: 1719-1724

[13] L. Lee, Fast context-free grammar parsing requires fast boolean matrix multiplication, *J. ACM* 49(1): 1-15 (2002)

[14] F. Magniez, C. Mathieu and A. Nayak, Recognizing well-parenthesized expressions in the streaming model, *STOC 2010*: 261-270

[15] G. Myers: Approximately Matching Context-Free Languages. Inf. Process. Lett. 54(2): 85-92 (1995)

[16] H. Samimi, M. Schaefer, S. Artzi, T. Millstein, F. Tip and L. Hendren, Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving. *Int'l Conf. Software Engineering 2012*

[17] L. Segoufin and C. Sirangelo, Constant-Memory Validation of Streaming XML Documents Against DTDs. *ICDT 2007*: 299-313

[18] L. Segoufin and V. Vianu, Validating Streaming XML Documents. *PODS 2002*: 53-64

[19] S. Staworko and J. Chomicki: Validity-Sensitive Querying of XML Databases. EDBT Workshops 2006: 164-177

[20] N. Suzuki: Finding an optimum edit script between an XML document and a DTD. SAC 2005: 647-653

[21] E. S. Tabanao, M. M. T. Rodrigo, M. C. Jadud: Predicting at-risk novice Java programmers through the analysis of online protocols. *ICER 2011*: 85-92

[22] A. Thomo, S. Venkatesh and Y. Y. Ye, Visibly Pushdown Transducers for Approximate Validation of Streaming XML. *FoIKS 2008*: 219-238