ON REPAIRING STRUCTURAL ISSUES IN SEMI-STRUCTURED DOCUMENTS

SHANSHAN YING

(Bachelor of Engineering)

EAST CHINA NORMAL UNIVERSITY, CHINA

THESIS

Supervisor: **ANTHONY K.H TUNG**

Department of Computer Science

School of Computing

National University of Singapore

2014

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

[illegible]

Shanshan Ying

12 Oct 2014

ACKNOWLEDGEMENTS

First, I would like to express my deepest thanks to my supervisor Prof. Anthony K. H. Tung. I sincerely appreciate his guidance, patience and encouragement, which helped me survive all challenges, pains and even desperation during the period of the candidature.

I am sincerely grateful to Prof. Zhenkai Liang and Prof. Wengfai Wong for their advices on my thesis. As my thesis advisory committee members, they both gave me valuable guidance from the very beginning of my Ph.D to the composition of my thesis. I am deeply appreciative of their suggestion on my thesis proposal, not only my on research, but also on the attitude towards research.

I also would like to thank Divesh Srivastava, Flip Korn and Bana Saha from AT&T Research. During my internship with them, I learnt a lot from them. They gave me invaluable suggestions on my work and impressed me with their love on doing research. I want to express my thanks to Dr. Srivastava and Prof. Beng Chin Ooi for providing me the chance of being an intern in AT&T labs. The six months' internship in New Jersey will be a memorable time in my life.

My senior fellows Su Chen, Xiaoyan Yang, Zhenjie Zhang, Dongxiang Zhang and Sai Wu, I am so appreciative to the help and care they gave to me on both my research and my life. They took care of me like my elder sisters and brothers.

Without their support, the Ph.D life would be tougher to me.

My colleagues and lovely junior fellows, Xiaoli Wang, Zhan Su, Feng Zhao, Jingbo Zhang, Meiyu Lu, Xuan Liu, Meihui Zhang, Yanyan Shen, Peng Lu, Feng Li, and everyone else in my lab, there are so many of you that I cannot name you all. Thank you all for your accompany. I will always remember the joyful and bitter days we spent together. Without you all, the Ph.D life would be quite boring!

I am always grateful to my long-term house-mate, Yuyao Cheng. We knew each other since we started our Ph.D in NUS, and we have been house-mates for almost six years. We made fun of each other, and we encouraged each other. We have gone through together the ups and downs during our Ph.D study. I am always appreciative for your tolerance on my bad temper and innocent behavior sometimes. Without your listening and comforting, it may take me a longer time to recover from the negative sentiment.

I also want to give my thanks to my undergraduate roommate, Ping Liang, Dan Wang, Zhenjie Wang, Xiaolu Li, Manmei Tang, Ni Tan and Liping Zhang. I can always find one of you to talk with, whenever I am feeling down. And I also want to thank my best friends, Yin Zhou and Jia Ying, though we are pursuing degree in different countries, we fight together; though we spent few days together during the past five to six years, we still feel free to talk to each other, and I am very grateful to life there are someone in the world understands me so well.

Last but not least, I would like to express my gratitude and love to my dear parents, and my little brother. My dear mum and dad, without your consistent support and love, I would definitely not be able to make it. You know little about what research is, and how to do research, but you respect whatever decision I make, and teach me to be patient and persistent. I know that I am easily losing my tamper when I feel stressed. So every time when I got into difficulties, I called you and you always know how to calm me down and give me the power to stand up and continue fighting. Your love is the strongest shield to me in the world. And my little brother, Xudong Ying, who also majors in computer science and is an NOIP gold medal winner, you gave me advices on algorithms and coding, and we shared the joys on practicing calligraphy. It is fabulous to have such a brother, to share the habits, and to understand the suffering and happiness of being a CSer.

CONTENTS

ABSTRACT

Poor quality of data can have a substantial social and economic impact. Although data quality management is a well-established research area, the vast majority of prior works focus on relational data. Increasingly, semi-structured data, such as XML and JSON, are becoming the de facto standard for a huge variety of data formats and applications. Their flexibility and easy-customization contribute to the soaring popularity of semi-structured data, but also serve as significant sources of major data quality errors. Well-formedness of structure, a prerequisite for many research works on semi-structured data, is an assumption often does not hold. Many XML documents suffer from erroneous structures, such as improper nesting where open- and close-tags are unmatched. Apart from this, tags are possibly organized in an incorrect hierarchy or sequence, leading to unexpected number of occurrence.

To enforce the balance of open- and close- tags, we propose in this thesis two algorithms targeting at different structural constraints. The first algorithm focuses on tags only while the second limits the occurrence of text in the document. Thorough proofs are presented on the completeness and approximation ratio of these algorithms. Besides we concentrate on detecting unexpected element error, when there are missing or spurious elements. We propose novel techniques to detect unexpected element errors and provide plausible reasoning for every reported error and a summarization technique based on variations of set cover for concise reporting. We demonstrate the effectiveness of these algorithms on real datasets through extensive experimental study.

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

INTRODUCTION

Poor data quality is a serious and costly problem that affects both traditional database and data on the web. Low quality data may cause significant loss for businesses (hundreds of billions of dollars per year) [36], and lead to low quality decisions. The most common data quality problems include missing data, incorrect values, duplications and inconsistency. These problems occur for a variety of reasons, such as incomplete information, weak integrity constraints, data integration from multiple sources, evolution of the schema, continuously changed data shoehorned into outdated schema, and erroneous input.

To enhance data quality, "anomalous" data causing low quality must be detected and repaired. The definition of "anomalous" is domain and application dependent. One data assumed anomalous may seem normal under another circumstance. Therefore, rules or constraints used to detect "anomalous" data should be general enough to capture the key characteristics. A large body of work from the database community has focused on this problem during the past two decades. For traditional database, there are works on data repairing, condition functional dependency inference, and for data on the web, there are works on entity resolution [33, 34] and duplication detection [35] techniques, etc. With the emerging of semi-structured data, there also emerge works on cleaning these documents, such as key inference [9], duplication detection [85, 83], consistency verification [26]. Such techniques cure the data quality problem from different aspects: normal form definition on designing schema and the semantic consistency.

Data cleaning has been under active research and attracts lots of attentions, from schema design, constraints discovery, to data repair, etc. Recently many more sophisticated data cleaning frameworks have been proposed and developed, such as LLUNATIC [45], NADEEF [32]. Few previous techniques, however, focus on the repairing the structural issues in semi-structured data.

Semi-structured data provide 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. Such flexibility wins it wide applications, in particular, the XML documents, since the advent of Internet. XML is the default format for many office products, such as MS Office, Open Office; is used for data storage for many datasets, such as Protein Sequence Database(PSD), Digital Bibliography Library Project(DBLP); is applied in data exchange, such as Rich Site Summary(RSS) feeds; and is even recommended to describe image, such as Scalable Vector Graphics(SVG). At the same time, this flexibility makes it more prone to errors. Data in traditional database are flat in structure but this does not hold in semi-structured data, which brings in new challenges. Techniques for relational database cleaning do not work perfectly here. One simple but widely spread structural error is: mismatched open and close tags, which is called *Tag-Level* error throughout the thesis. Though some mismatches, e.g., in HTML, can be repaired by the browsers parser, not all the browsers repair in the same way, leading to inconsistent display. One open tag and its matching close tag are the base of an element, and generally an element could contain a list of other elements or some texts. By examining elements, we observe some *Element-level* errors: unexpected number of occurrence of elements. Unexpected element errors refer to the presence of spurious elements or absence of required elements. For example, in the DBLP dataset, people may edit one `<inproceedings>` by inserting some `<editor>`, instead of `<author>`, which is incorrect in semantic but will be accepted by the schema of DBLP. Existing works on cleaning semi-structured data do not consider such structural issues and assume the input free from structural errors. To improve the quality of such documents, in this thesis, we focus on data cleaning on semi-structured documents against these two levels of errors.

1.1 Mismatched Tags Repair

A recent study of XML documents on the Web found that 14.6% of them (out of a 180K sample) are not well-formed, the majority of cases due to either open- and close-tag mismatches or missing tags [49]. A Google study in 2005 on (XML-based) RSS feeds on the Web found that 7% have some errors, the largest kind (after non-compliant UTF-8 characters) being open and close tag mismatches [1]. Such errors are due to multiple factors including manual input [75], dynamically-generated data from faulty scripts [69], 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).

[1] `http://googlereader.blogspot.sg/2005/12/xml-errors-in-feeds.html`

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 [49]. Inferring one is a notoriously difficult problem [22], often requiring a whole repository rather than a single document, and which for some classes of documents is not even possible [47]. Therefore, most existing work assumes that the document is well-formed and tests validity based on a supplied grammar [72, 71, 24]; exceptions to this include `HTML Tidy` and `NekoHTML`, both of which are specifically tailored for HTML documents.

We first consider the problem of repairing an arbitrary semi-structured document into one that is well-formed, based on two variants of well-formedness. We believe this problem is in itself interesting for a variety of reasons. First, some existing documents have a very flexible grammar that basically requires only 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 auto-correction.

While *verifying* well-formedness in semi-structured data can be done in a straight forward way, 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). Another commonly used

```
<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 1.1: An Example and Rule-based Repair

rule is to insert a matching close-tag whenever the current close-tag does not match, but this can trigger a similar cascade.

Example 1.1 Figure 1.1(a) shows 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; the `</title>` close tag is out of order, occurring after `<authors>`; etc. Figure 1.1(b) shows the document after the substitution rule-based heuristic is applied, requiring 3 substitutions and 2 insertions.

We focus on the following types of errors that we believe occur most frequently in practice:

- Tags may be missing, as it is common to forget to close open tags, and unmatched close tags may occur when new content is added and it is assumed a previous open tag existed.
- Extraneous tags may be present, perhaps due to not fully deleting tags associated with deleted content.
- Open and close tags, due to being similar, are sometimes mistaken for each other; and tags of different types may appear in the wrong order or be improperly assigned.

We use standard string edit distance with insertion, deletion and substitution operators as a model for repair [58]. We believe that more complex distance functions including other operations, such as block moves and swaps, as well as non-uniform weighting, can be folded into our methods but we leave this to future work. [2] Edit distance is used for modeling and correcting errors in many applications from information retrieval to computational biology [79, 65].

One limitation in data repairs work is we never know what is absent from the data, and what is the true value of a dirty data. Repairing the data towards the most possible or reasonable direction is the thumb of rule. The most widely accepted norm is to repair data with as little cost as possible. Therefore, we use minimal edit distance as the target under the theme of finding minimal or lowest cost changes to the data that make it consistent with the constraints.

[2]While additional operations such as swaps and block moves would certainly enhance the model for some scenarios, considering them greatly complicates things.

```
<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 1.2: Two Possible Repairs

Such theme has been widely adopted in almost all the data cleaning work for inconsistency repair [31] and constraints repairs [52, 23] and many others.

In our illustrative example, a well-formed repair with fewest edits is given in Figure 1.2(a), which has edit distance 2: delete `<authors>` and delete `<affiliation>`.

In our second variant of well-formedness, we take into account that the text embedded within semi-structured documents often follows certain patterns. For example, most XML documents only allow text to occur surrounded by matching open-close tags and require the existence of text between every adjacent matching pair. Thus we consider how to exploit embedded text to aid in finding a more judicious repair via a constrained edit distance function. In our illustrative example, a well-formed repair based on tags and text with edit distance 3 is given in Figure 1.2(b): delete `<authors>`, insert `<affiliation>` before `IBM`, and substitute `<affiliation>` after `IBM` by `</affiliation>`. Note that this repair consists of more edits than for tags only.

Note that it is not always possible to exactly repair to the originally intended well-formed string. In the absence of a grammar, there is inherent ambiguity in what the creator intended. For example, consider the string `<name> E. F. Codd </author>`. Should this be repaired to `<name> E. F. Codd </name>` or `<author> E. F. Codd </author>`? Or even to `<name> E. F. </name> <author> Codd </author>`. It is impossible to know what the original intent was. Furthermore, such ambiguities compound in larger strings, resulting in an explosive number of reasonable possibilities. Since the user may have a (often ill-defined) grammar in mind, our methods can provide multiple repairs in the hope that at least one of these will suffice. But presenting the user with all repairs based on the many ways to resolve these ambiguities can be overwhelming. Instead, we note that the differences between some repairs are syntactically trivial, so we try to consolidate these into representative repairs. For example among the two alternatives `<name> E. F. Codd </name>` and `<author> E. F. Codd </author>` to repair `<name> E. F. Codd </author>`, we canonically choose the former.

Consolidating multiple repairs by such representatives helps to provide more variety in a small set of repairs returned to the user. For the second variant(Tags With Text case), the surrounding text can be exploited to resolve more of these ambiguities. For example, if from a well-formed string such as `<name> E. F. Codd </name>` a tag gets deleted resulting in `<name> E. F. Codd` or `E. F.`

`Codd </name>`, indeed our algorithm will repair it by inserting the deleted tag. Therefore, with a stronger grammar, there are more cues to recover the original string.

There has been much literature on approximate matching of trees which has been applied to finding semantically relevant XML documents [30, 50, 80, 68, 86]. Unfortunately, none of this work applies to our setting since the input is not well-formed and, therefore, cannot be represented as a tree. However, a good repair should result in a short tree edit distance between the repaired string and the intended error-free string. We use this to show the efficacy of our algorithms in "undoing" errors introduced to a well-formed string. Recall that for the reasons of ambiguity mentioned above, it is not enough to simply check whether or not the repaired string is the *same as* the intended error-free string. In addition, our experimental evaluation on real XML data with real errors shows that the number of string edit operations is much smaller when using our approach compared to the rule-based heuristics. *This effectively establishes the goodness of edit distance for repair.*

1.2 Unexpected Elements Detection

Documents with proper nesting are called well-formed. But well-formedness is just the beginning, not the end of story. Using string edit distance as the metric, it is possible that the repair is far from user's intention, with elements nested in an unexpected order, or missed, etc. With these anomalies (elements with unexpected number of occurrence) detected, the quality of repairs could be further improved. Similar observation could be made from many online XM-L documents, mainly maintained manually. A recent study [49] reveals even when documents are well-formed, many of those are invalid due to *unexpected element errors*. Unexpected element errors refer to the presence of spurious elements or absence of required elements. For example, in the DBLP dataset, we detect several `articles` with duplicated `title` elements, or missing the `journal` name in which they appear. Some of them misuse `editor` tag to indicate `author` etc. The existence of these errors leads to poor performance on basic queries over the underlying data [77]. Even worse, it may result in incorrect answers, and false decision making. While prior works have considered automated repairing of malformed documents to make them well-formed [53], and to check validity of documents based on schemas–these works are not suitable for our purpose. In this work, we go beyond well-formedness and validity, and propose novel techniques to handle structural anomalies due to unexpected elements.

The foremost question that we need to answer is *what constitutes an unexpected element*. Schemas, such as DTDs or XSDs for XML documents, use quantifier to restrict the number of occurrences of a particular element. Since schemas are often designed manually and meant to be easily readable, they are often over-simplified. Therefore, even when a document is valid according to a schema, the possibility of an unexpected element error cannot be ruled out.

Example 1.2 Consider a toy XML example in Figure 1.3 describing the political divisions of countries. The semi-structured document is parsed into a document tree, with a single root node `countries`. Each node in the tree under `countries` corresponds to an element in the document. All attribute values and some attribute names are omitted here for simplicity.

Upon seeing the three left-most entries of `country`, one may use the following production to define the sub-elements under a country node:

$$country \rightarrow name\ [province|city|state]^*.$$

According to this rule a country node should have a single name followed by zero or more occurrences of province, city and state. However, a country may not have both state and province. This is not captured by the proposed schema, and as such, the fourth entry (in dashed rectangle) though erroneous, appears valid according to the schema.

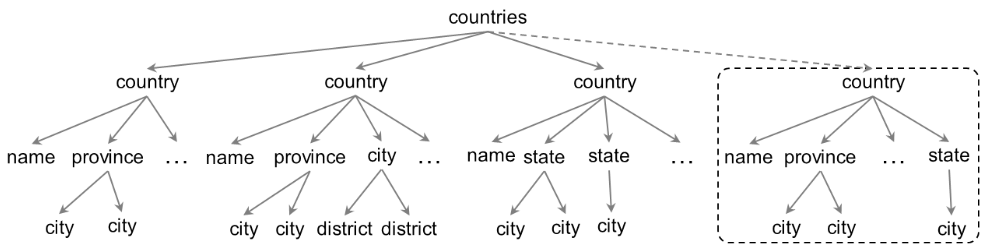


Figure 1.3: An XML Document Example

The above example also illustrates that as new data arrives over time, it is possible for a schema to become obsolete. Discovering structural anomalies based on an obsolete schema may lead to both high rates of false-positives and false-negatives. For example, as more data is inserted into the document in Figure 1.3, some of the countries may have multiple names associated with them: United States of America, USA, the States and America all refer to the same country. Any city directly under country may need to be renamed as province etc. Inferring a valid schema and adjusting it timely with updates is a hard problem. All existing works on schema inference assume data to be clean [19, 21, 18]. Therefore these techniques do not lend themselves to structural error detection. The inferred schema suffers from over-fitting and is often hard to read owing to large size. In addition, the number of documents with available schemas are low. Among the 180,000 semi-structured documents collected in [49], only about 24% have accompanying schemas. Hence relying on schema definition alone is inadequate to discover structural anomalies, the like we consider here.

An alternate approach is to use the data statistics directly, that is *letting the data to speak for itself*. In some sense, we want to identify occurrences/non-occurrences that are not observant of expected behaviors. However, it may be tricky to mark occurrences as rare. We illustrate this using an example.

Example 1.3 Suppose we count for each `country` element the number of sub-elements labeled as `province` to get the expected number of occurrences of `province` in Figure 1.3. After visiting all `country` nodes, we get out of a total of 200 countries, 150 have two provinces, 45 have 1 and 5 have 0 provinces. Therefore, we get a percentage distribution as $\{2.5\%, 22.5\%, 75\%\}$ for having 0, 1 or 2 provinces. Suppose we set a relative threshold of 3%, indicating the number of countries having 0 province is below the set threshold accounting them as errors. It may turn out that all these 5 countries have `state` underneath them, and having 0 province is perfectly valid under such circumstance. On the other hand, there are another 5 countries with 1 province each which also have a `state` node underneath them. In fact, these are the true errors: both `province`, and `state` cannot coexist under country. This method of finding relative frequency to identify rare events therefore detected 5 false-positives and missed 5 true errors (false-negatives).

In the above example, true errors can be detected if we look at the conditional distribution of `state` under `country` with child `province`. However, exploring arbitrary conditional distribution is computationally infeasible. Moreover, a good error detection mechanism must also provide justification for reporting an element erroneous. Considering arbitrary conditional distributions suffers from the obvious drawback that providing any comprehensible explanation for reported errors soon becomes prohibitive.

The `country` in the above example serves as the *context* for calculating the relative frequencies of `province`. Such context specific mining of conditional distributions is very important, and is in the heart of our techniques. For example, it is possible for a `city` under `country` to have `districts`, but a `city` under `province` cannot.

1.3 Contributions of the Thesis

In this thesis, we study the data cleaning problem in semi-structured document by investigating different repair constraints where the structural errors could be detected, from the aspects of tag-level and element-level. For the tag-level errors, we show two constraints to repair against, and propose several algorithms efficiently solve this problem. For the element-level errors, we put forward the definition of *Explanation* to detect the elements with unexpected number of occurrence under certain circumstance. In particular, our contributions are as follows.

We study the tag-level errors, where there are some open- or close-tag missing. There are two variants of its kind: tag-only and tag-with-text. To solve these problems, we give a dynamic programming algorithm which computes the optimal edit distance in $O(n^3)$ time, independent of the grammar size. Since this algorithm is cubic in the size of the input, it does not scale to large documents. We also propose branch-and-bound algorithms when multiple repairs are desired (such as for an auto-correction menu), since the dynamic program and greedy algorithms are geared towards 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 thorough experimental study to investigate these strategies on real data.

We then study the element-level errors. As far as we know, we are the first studying the conditional number of occurrence of elements in semi-structured documents. We formally define *Explanation* as a triplet to encode the conditional distribution and then propose the way to organize these explanations in a lattice for each target tag to capture as many anomalies as possible. Finally we use a greedy algorithm to do a summarization. Extensive experiments are done on several real datasets, and a visualization tool is developed for a better interactive repair.

1.4 Outline of the Thesis

The remainder of the thesis is organized as follows. In Chapter 2, we present literature review on existing techniques on semi-structured documents verification and key inference. From Chapters 3–5, we propose repairing errors and identifying errors of different levels. Chapter 3 presents solution for documents when only open- and close- tags should be matched and proposes algorithms to satisfy various demands. Chapter 4 introduces a more restrict constraint, where each text must be surrounded with a pair of tags and each matching pair should have either text or child tags. Chapter 5 presents the problem on detecting anomalous elements with unexpected number of occurrence, and how to get a concise summarization to explain these anomalies. Chapter 6 concludes the thesis and lists some future work to improve the quality of semi-structured documents.

Chapter 2

LITERATURE REVIEW

Tremendous work have been done on semi-structured documents during the past decades, ranging from schema design, keywords query, to constraint inference and duplication detection. In this chapter we first review key techniques contributing to semi-structured documents repair, verification, as well as key and schema inference and schema repair, and then introduce some techniques on data summarization for query results.

2.1 Document Repair and Verification

2.1.1 Well-formedness Repair

While we are not aware of prior work that specifically addresses the problem of repairing malformed semi-structured document to make it syntactically well-formed, there is some work on repairing XML documents to make them valid with respect to a given DTD [24, 73, 74, 76], by recording possible state transition information for each node in the automaton. However, these papers all assume the input is already well-formed and DTD can be formalized as a tree structured where no self-recursive exists. 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` [2], `Html Tidy` [3] and `NekoHTML` [4] allow for malformed HTML input and exploit pre-defined domain knowledge to make them valid; however, they are specially tailored for HTML documents and not work well for an arbitrary input, as they use rule based algorithm to fix unmatched tags.

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(Context Free Grammar), these existing solutions can be applied. Aho and Peterson [8] gave an $O(|G|^2n^3)$ algorithm which was later improved to $O(|G|n^3)$ by Myers [64],where n is the length of the input and $|G| = \sum_{A\to\alpha\in G}(|\alpha|+1)$ is the size of the grammar. For context-free grammars, which includes well-formed bracketed expressions (also known as a Dyck language), a $O(|G|n^3)$ algorithm based on CYK parsing exists [64]. For regular grammars, which are not powerful enough to capture bracket languages, an $O(mn)$ algorithm exists, where m is the size of the regular expression.

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 [48]. It is also known that parsing an arbitrary CFG is at least as hard as boolean matrix multiplication [56]. Therefore, computing the edit distance to a well-formed string in much less than cubic time would be a significant accomplishment.

2.1.2 Constraints Verification

Verifying well-formedness is a 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 sub-linear space [62]. 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. Static verification can be done by walking through the tree automata(which models the DTD), and verifying either in a BFS or DFS way depending the underlying parser(SAX or DOM). To support incremental validation, auxiliary structured record the states each tag belongs to to speed up transition. So that deletion, insertion and update can be supported by checking a handful of tags. Some of these papers (e.g., [71]) perform *strong validation*,checking for well-formedness along with validity,while others (e.g., [14, 67, 13, 63]) perform *weak validation*, assuming the input is already well-formed.

Our work fits into the context of data cleaning to satisfy database integrity constraints, including consistency under functional dependencies [16],inclusion dependencies [23] and record matching [40]. All these works can be generally modeled into following problem: repairing the data D to satisfy certain constraint T where the repaired data D' has minimal distance $Dist(D, D')$. Though the exact definition of unit cost sometimes differs from applications, most of them use edit distance as a notion of a minimal cost repair. Hence in our well-formedness repair problems, we also take minimal edit distance as the metric to be optimized.

2.2 XML Constraints and its Inference

2.2.1 XML Constraints

Generally there are two kinds of constraints associated with one XML document one defining the structural constraint and the other for semantic constraint, respectively. Structured constraint, limiting the tag nesting and number of occurrence, is usually represented as a *DTD*(Document Type Definition) or an *XSD*(Xml Schema Definition). Many existing works propose various languages and models in defining the structures though, DTD and XSD are still the mainstream.

These two constraints are rarely applied side by side. The hardness lies in the proof of consistence between these two types of constraints. As proved in [11, 10], the problem of proving consistent between the semantic constraints and structural constraints is NP-hard. Hence, structural constraint and semantic constraints are studied independently to reduce the complexity. Most importantly in real life, people who consult to XML as the data storage model, are attracted by its convenience of flexible grammar, and will not have so many constraints to be meet at the same time.

Semantic constraints refer to integrity constraints on XML. Constrains such as keys, functional dependency and satisfactory have been deeply investigated. Readers may refer to [37] and [26] for more detail for more discussion on constraints of this kind. In the following, we specifically introduce two kinds of semantic constraints that are proposed recently.

2.2.1.1 Cardinality Constraint

The min- and max-occurrence of elements can be declared in XSD clearly, which restricts structural occurrence. Cardinality constraint, proposed by Link et al. in [42, 41], focuses on data semantics. Cardinality constraints are defined in terms of path expression, and restrict the number of elements that have the same values on some selected sub-trees [61]. In other words cardinality constraints capture information about the frequency with which certain data items occur in particular contexts.

With cardinality constraints posed, it restricts the cardinality of the answer to some query against the dataset and can help estimate the selectivity of the query. Such estimation is useful, for instance, when users issue q query through mobile phone and the network costs is essential to decide continue the query or abort. In [41] Ferrarotti et al. study the implication and compatibility for a given set of rules and prove the complexity of the problems to be co-NP hard. Then they propose an efficient algorithm for deciding implication. A new class of constraint, Soft Cardinality Constraint is designed, which needs to be satisfied on average only, and thus permit violations in a controlled manner.

Considering the description complexity of the constraints, understand a dataset and construct proper cardinality constraints are quite challenging to user or database administrator. In most cases, such constraints are less intuitive. However, all exiting works focus on how to deal with the constraints when you have one, rather than how to discover one.

2.2.1.2 XML Functional Dependency

The concept of Functional Dependency origins from relation database, posing on constraint on two attributes a_i, a_j in a relation or a table R. Attribute a_i is said functionally determines a_j if each a_i value is associated with precisely one a_j value.

Similar dependency is observed in semi-structured data. Lee et. al. are the first proposing designing XML Functional Dependency(FD) in [57] and later Libkin gives formal definition and shows normal forms of XML FD in [12, 60], by defining the notion of `tree tuple`. [78] puts forward another view of normal forms, using a path based approach, using a set of paths to identify the condition elements.

Both methods effectively capture multi-hierarchical constraints, but are far from satisfactory. Neither works effectively when dependent elements are a set elements. For instance, it is more reasonable to claim `ISBN` of a book determines a set of `author`, not a single `author` element. To make up the flaw, Yu improves the definition of `tree tuple` [85, 83], and shows the advantages in capturing more constraints using `Generalized Tree Tuple`.

The best way to define a Functional Dependency Constraint is still under active discussion. There is still no standard in the field on defining XML Normal Form, based on XML FDs or XML Keys.

2.2.2 Schema Inference

Observing the crucial fact that 1) Many XML documents do not have any schema (as high as 25%), 2) the schema may be out of data(about one third of the documents failed to meet the satisfactory constraints), it is helpful to infer concise schema from the given set of XML documents. With the definition of schema differs, the algorithms changes. The detail is presented as followings.

2.2.2.1 Inference of DTD

Several approaches have been proposed for DTD inference. XTract [44, 43] generates a set of candidate regular expressions from each element. As DTD is unaware of context, label of an elements solely define the type of an element. Following the Minimum Description Length principle, the most concise one is selected as the base answer. [70] uses several approaches to generate probabilistic string automata representing regular expressions, by application of inductive inference theory. But the inferred result is automata, and there is no conversion to the standard DTD, or regular expression.

In contrast, Geert et al [19] propose to infer concise DTD*phDocument Type Definition* from the XML data. As DTD is context free, it can be converted to equivalent RE(*Regular Expressions*). So they consider two RE types that can cover more than 95% of the DTDs: *single occurrence regular expressions*, in which every element name can occur at most once, and *chain regular expressions*, which is a chain of expressions, and there is no quantifiers $(?, *, +)$ within each expression. The core idea consists of three steps. First, construct an automaton from the input positive documents. Second, convert the automaton to regular expression using a set of heuristics to induce disjunction, concatenation, self-loop, etc., relationships between automaton nodes. Third, apply the heuristics until one path in the automaton. To infer the chain regular expression, they constructed a simple directed graph which is loop-free with single occurrence, so as to derive the RE more directly. As the DTD in real life is supposed to be simple, their simplified algorithm resulted in good efficiency and accuracy. User may refer to [20] for more details.

2.2.2.2 Inference of XSD

XSD (*XML Schema Definition*) is another widely used schema in XML, up to 65% of the documents with schema is associated with an XSD. The major difference between a DTD and XSD is: the latter is context aware while the former is not. Given one XML document about NBA players and teams, `<players> <player> <name> <firstName> Kobe </firstName> <lastName> Bryant </lastName> </name> </player> ...</players> <teams> <team> <name> L.A. Lakers </name> </team>...</teams>`, DTD cannot tell the difference that the tag `<name>` under `<player>` and `<team>` are different, while an XSD can easily distinguish the disparity by setting constraints on path of the tag. We can simply view the difference as, in DTD, the type of tags are defined globally, while in XSD, types can be defined locally.

To infer the context-aware regular expression, Geert et al [22] introduced a new parameter:k,length of xml path, as the context of each tag. Regular expressions that are similar to each other are generalized into one to reduce the rule space. The core algorithm here is two-step: generate the RE with varied path with k nodes and merge rules exceeding a given similarity threshold. This is work is heavily dependent on their previous work on inferring DTD. The functionality of XSD inference is also embedded in many softwares. Trang [5] is the state-of-the-art software designed for schema translation between DTD, XSD, etc, and for schema inference for a given set of XML documents. However, the generated XSD is context independent, which makes the inferred XSD no different from DTD. Microsoft .NET framework [7] and XMLBeans [6] also provide the XSD inference tools. But the one from .NET does not work for nested structured, and the one from XMLBean suffers from the same problem as Trang.

Recently there are few new findings in inferring schema, which is suffering from following drawbacks: First, we can only learn what is present from the positive documents and never know what is absent. Second, the inferred schema, represented as regular expression, is not readable or user-friendly, especially when the tags space is large. Users can hardly harness the inferred schema to understand the document set. Third, the schema inference works only for clean data, which is rarely the case in reality, as more than 30% documents with schema do not comply with the schema. Therefore, it would be of great help to infer a set of rules, which are context aware and reflect the interestingness of tags under different context.

2.2.2.3 Inference of Key

In there context of traditional database, a fundamental of works focus on data inconsistencies, by inferring FD(Functional Dependency)[51], CFD(conditional functional dependency)[27] and repairing documents[28, 17] w.r.t CFD, etc. The number of works on XML integrity constraints is relatively much smaller.

A very recent work from Marcelo et al. infers keys [9] with the presents of schema. Keys, in XML, like CFD, are context aware. Taking the previous N-BA players and teams document for example, `<name>` tag can uniquely identify one NBA team, as there is no duplication here, but not the <name> tag under <player>. They evaluate the quality of a key from four universality, boundedness, key implication and satisfiability and they characterize the difficulty for each of them with thorough proof.

2.3 Data Summarization

Data summarization serves users a more concise result set with high accuracy, and is especially important when the original data much too complex to understand or the data size is overwhelming. Data summarization has been an active research area, and has covered almost various types of data, from database query result [54, 25], data cube semantics [55], relational database schema [81, 82], to graph data [87, 66], and XML schema summarization [84].

Though differ in detailed techniques, the essence of data summarization is the same: concise and accurate. Concise limits the size of the data, and accurate requires the summarization to cover as much information as needed. Generally, a good summarization should blur those less important information and highlight the interesting parts, which could be distribution, connection structure, etc. Certain inaccuracy is allowed for a more concise summary, and the measure of accuracy is highly dependent on applications. A widely standard metric for conciseness is Minimum Description Length (MDL) principle, proposed by Rissanen [15].

[54] and [25] are the two most relevant works to ours, considering applying MLD to summarize a set of query result, where the result are associated with some hierarchies. Given a set of result set, marked in different color: blue (must cover), red (never cover), and white(do not care). In [54] Lakshmanan proposes

a generalized MDL approach for summarization, where `generalized` refers to the permission of bounded number of white cells. Two cases are studied: the spatial case when there is no hierarchy to restrict the formation of regions, and the hierarchical cases, which is NP-hard when there are more than one hierarchy hitting the same set of data. [25] solves the summarization problem from another perspective, by allowing "holes" or exceptions, called MDLH So the total description length is the number of regions, plus the number of exception cells. Several summarization algorithms are designed, based on greedy, dynamic programming and quadratic programming, but none is optimal. The MDLH approach offers a good summarization easier for users to understand, when the number of exceptions are small. The major difference between this work and ours is: there is no overlap among sibling nodes in the hierarchies. When nodes are disjoint, the one-dimension case can be solved in PTIME, but such algorithm does not work when overlap occurs.

Chapter 3

REPAIR WITH TAG-MATCHING CONSTRAINT

With the absence of an accompanying schema, tag well-formedness is the constraint one document must comply to. In this chapter, we focus on computing syntactic repairs against a malformed input document, under the constraint that tags must present either in matching pairs or nicely nested. We propose a dynamic programming algorithm and a branch-and-bound algorithm targeting for various user demands, and experimentally demonstrate the efficiency and accuracy of these algorithms on two real datasets.

3.1 Motivation

Driven by the eagerness of information exchange over the internet, semi-structured documents have won a wide range of applications in various domains. XML (eXtensible Markup Language), email, EDI(Electronic data interchange) are representative models of its kind. Thanks to its flexible grammar, even ordinary users can create such documents with minimal training. However many semi-structured documents are suffering from non-wellformed errors. Taking the XML documents for instance, Grjzenhout et al. report in [49] that either open- and close- tags mismatching is the major contributor leading to erroneous XML documents, and mis-matchings could be caused by manual editing, conversion, or by buggy program, etc.

While there are many works focusing on inferring schemas, removing redundancies, validating checking, little attention is paid on the structural problems of these documents. Most existing works assume the input document to be well-formed, which is a little bit strong. As a prerequisite of many other works, how to efficient repair these documents whit the absence of schema is by itself an interesting problem.

3.2 Problem Definition

In this chapter we focus on how to repair the document on tags. We shall ignore all other components in the XML document besides tags such as attributes and text and treat each self-closed tag as two tags, e.g., <a/> becomes <a></a>. We assume that the input document is tokenizable by some 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.

Definition 3.1 Congruent

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 (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 3.2 Matching Brackets

A *match* between two brackets x and y, denoted $x \asymp y$, occurs when $x \in T$, $y \in \bar{T}$ and $y = \bar{x}$. x and y form one pair of matching brackets.

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

Definition 3.3 Well-formed String

A *well-formed string* s over some bracket alphabet T obeys the context-free grammar G_T with productions $S \to SS$, $S \to \varepsilon$ and $S \to xS\bar{x}$ for all $x \in T$. [1] So $|G_T| = 4|T| + 3$.

[1] 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.

Example 3.1 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 3.4 Well-formed Bracketed Language

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 3.5 String Edit Distance

The *edit distance*, denoted as $E(s, s')$ 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 $s_1...s_is_{i+1}...s_n$ to $s_1...s_ias_{i+1}...s_n$; a deletion at position i transforms $s_1...s_{i-1}s_is_{i+1}...s_n$ to $s_1...s_{i-1}s_{i+1}...s_n$; and a substitution to a at position i transforms $s_1...s_{i-1}s_is_{i+1}...s_n$ to $s_1...s_{i-1}as_{i+1}...s_n$.

Now the problem of repairing one non-well-formed document can be formalized as fixing a sequence of brackets to satisfy the Well-formed Bracketed Language with minimum edits.

Definition 3.6 Bracketed Language Edit Distance Problem

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 refer to the edit distance from the string to a well-formed repair.

Example 3.2 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.

3.3 An Optimal Solution using Dynamic Programming

In this section we propose a dynamic programming algorithm to solve the problem and prove the optimality of the algorithm, followed by pruning strategy to speed it up.

3.3.1 Dynamic Programming Algorithm

According to Definition 3.3, given some substring $s_i...s_j$, where $j > i$, either s_i and s_j could be edited to matching brackets (the $S \to xS\bar{x}$ production) or the string could be broken into two adjacent well-formed substrings (the $S \to SS$ production). Let $B[i,j]$ be the cost of editing s_i and s_j to match, and $C[i,j]$ the edit distance to repair substring $s_i...s_j$. Then the recurrence is

$$c[i,j] = min \begin{cases} B[i,j] + C[i+1,j-1], & i < j \\ \min_{i \le k \le j-1} C[i,k] + C[k+1,j], & i < j \\ 1, & i = j \end{cases} \tag{3.1}$$

,where the cost $B[i,j]$ of editing s_i with s_j is:

$$B[i,j] = \begin{cases} 0, & \text{if } match(s[i], s[j]) \\ 2, & \text{if } s[i] \in \bar{T} \text{ and } s[j] \in T \\ 1, & \text{otherwise} \end{cases} \tag{3.2}$$

The intuition is that when the two brackets form a match, the cost is 0; when $s[i] \in \bar{T}$ and $s[j] \in T$, at least two substitutions or two deletions are needed; otherwise, the cost would be one, substitute either of the bracket to generate a match.

Algorithm 3.1: Dynamic Programming for Tag-Only

```
  Input: tokenized String $s = s_1...s_n$
  Output: Edit Distance $\min_{s'} E(s, s')$ where $s'$ is well-formed
1 forall the $\ell$ from 0 to $n-1$ do
2     forall the $i$ 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         forall the $k$ from $i$ to $j-1$ do
6             $C[i,j] \leftarrow \min(C[i,j], C[i,k] + C[k+1,j])$;
7 return $C[1,n]$
```

We adopt idea of dynamic programming(*DP*) to solve the problem. The pseudocode to compute the edit distance $\min_{s'} E(s, s')$ to a well-formed string s' is presented in Algorithm 3.1. The DP algorithm runs in $O(n^3)$ time and requires $O(n^2)$ space, where n is the string length.

Finally, $C[1,n]$ is returned as the edit distance. The following claim establishes the correctness of the recursion and hence, also of Algorithm 3.1.

Theorem 3.7 Given string s, the dynamic programming algorithm described in Algorithm 3.1 correctly finds the edit distance, such that the repaired string is accepted by G_T.

Proof. Suppose l is the length of substring s, we prove Theorem 3.7 using induction on l. $l = 1$: consider any substring in the form s_i. The minimum edit distance is 1, which can be achieved either by deleting s_i(using rule $S \to \epsilon$), or by inserting a matching bracket.

$l = 2$: consider any substring represented as $s_i s_{i+1}$. According to our definition, $B[i, i+1]$ is the cost paid to match s_i with s_{i+1}, so $C[i, i+1]$ equals to $B[i, i+1]$, which is correctly computed.

$l = k + 1$: We prove how the algorithm work for substrings with length greater than 2. By induction hypothesis, Algorithm 3.1 correctly computes minimum edit distance for all substring of length no greater than k. Take a substring of length $k+1$, represented as $s = s_m, ..., s_{m+k+1}$ without loss of generality. If the computed edit distance is global optimal, taking s_m as example, the optimal algorithm must cover following options:

- *Delete* s_m *and the edit distance is* $1 + C[m+1, m+k+1]$.

  The dynamic programming algorithm in Algorithm 3.1 covers this option, the algorithm has the choice to set edit distance as $C[m, m+k+1] = C[m, m] + C[m+1, m+k+1]$, and both terms on the right hand side are computed correctly by the algorithm according to base cased and induction hypothesis.

- *Match* s_m *to some other bracket* s_j *from the string, where* $j > m$.

  In this scenario, four subcases must be covered to guarantee the optimality, and we will discuss case by case in the following.

  (a) $j = m + k + 1$ and $s_m \asymp s_{m+k+1}$. In this case, minimum edit distance is $C[m, m+k+1] = C[m+1, m+k]$. The dynamic algorithm in this case computes $C[m, m+k+1] = B[m, m+k+1] + C[m+1, m+k]$. Since $B[m, m+k+1] = 0$according to Equation 3.2, the algorithm correctly computes the edit distance $C[m+1, m+k+1]$ by induction hypothesis.

  (b) $j = m + K + 1$, and s_m(resp. s_j) is substituted to match s_j(resp. s_m). If both s_m and s_j are close(resp. open) brackets. In this case, the minimum edit distance is $C[m, m+k+1] = 1 + C[m+1, m+k]$. The dynamic programming algorithm in this case gives a cost of $B[m, m+k+1] + C[m+1, m+k] = 1 + C[m+1, m+k]$, according to case 3 in Equation 3.2 and case 1 in Equation 3.2. By induction hypothesis, the second term in the right hand side is computed correctly.

(c) $j = m + k + 1$, $s_m \in \bar{T}$ and $s_j \in T$. In this case, matching s_m with s_j requires substitution or deleting both brackets, at a cost of 2, i.e.,$C[m, m + k + 1] = 2 + C[m + 1, m + k]$. According to the algorithm, it computes the cost as $C[m, m + k + 1] = B[m, m + k + 1] + C[m + 1, m + k]$ as one of the choice and $B[m, m + k + 1]$ is computed as 2, defined in Equation 3.1 case 2. By induction hypothesis, the second term in right hand side is computed correctly.

(d) $m < j < m+k+1$. By matching s_m with some internal bracket s_j in the substring, the minimum edit distance is $C[m, m + k + 1] = C[m, j]+C[m+j+1, m+k+1]$. From the dynamic programming algorithm, we can see that Equation 3.1 case 2 considers this subcase and the terms in right hand side are correctly computed by induction hypothesis.

The cases listed above are exhaustive, and Algorithm 3.1 covers all the possibility, computing edit distance correctly in all these cases and returns the minimum as the final result. Hence, Algorithm 3.1 figures out the minimum edit distance for any substring of length $l + 1$ exactly as we expected. Therefor we claim that 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 3.4.2.

3.3.2 Well-formed Substring Removal

The cubic growth in running time as a function of string size becomes a problem for large strings, especially when in real world most documents has tens of thousands of tags but relatively small in repair cost. Such observation reveals that even though the input s is not wellformed, it may consist of multiple discrete well-formed substrings $\{s_1^{wf}, s_2^{wf}...\}$. Ideally these substrings should be left untouched and be excluded from the algorithm to save time, but will it affect the edit cost? The answer is positive and will be proved in the following section.

Using a stack, well-formed substring removal can be done straightforwardly in linear time by recursively finding matching adjacent pairs and removing them. We note that this may eliminate some candidate repairs from consideration. For example, given the string $s = a\bar{a}\bar{b}\bar{a}$, the pruned string is $s^r = \bar{b}\bar{a}$, where $s_1^{wf} = a\bar{a}$. The only optimal repair on s^r can be found is $a\bar{a}a\bar{a}$, since s_1^{wf} will not be disturbed. Whereas the repair $ab\bar{b}\bar{a}$ 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. That is, the removal probably reduces the variety of repaired string, it will not affect the number of edits.

Theorem 3.8 Well-formed substrings removal preserves the edit distance.

Proof. Denoted as $Cost(s) = d$ the edit cost returned by Algorithm 3.1 to repair the string s. We do an induction on the number of edits d to prove that $Cost(s) = Cost(s^r)$, where s^r is the string after removing well-formed substrings from s.

$Cost(s) = 0$: For strings with $Cost(s) = 0$, well-formed substring pruning returns the empty string s^r, where $|s^r| = 0$, so its edit cost $Cost(s^r) = 0$. This serves as the basis.

$Cost(s) = d, d > 0$: Suppose the claim is true for all strings with minimum edit distance less than d, we check the correctness of the claim when $Cost(s) = (d)$. Consider an optimal algorithm that defers doing the first edit as much as possible without affecting the optimality. Let $\mathcal{P}$ and $\mathcal{P}^r$ be edit scripts set for s and s^r, respectively. , and let the d edit positions be $p_j[1] < p_j[2] < ... < p_j[d]$ and $p^r_k[1] < p^r_k[2] < ... < p^r_k[d]$ for edit script $p_j \in \mathcal{P}$ and $p^r_k \in \mathcal{P}^r$ respectively. Suppose the prefix in s^r corresponds to $s_1 s_2 ... s_{p_j[1]}$ of edit script p_j be $s^r_1 s^r_2 ... s^r_{q_j}$, which must be a run of open brackets, otherwise it breaks the assumption that $p_j[1]$ is the position of the first edit. Therefore whether or not $s_{p_j[1]}$ is part of some well-formed substring, we present that the claim hold in either case. When the answer is positive, we prove by reducing the two strings to the same pruned string; when the answer is negative, we show there must be an alternative edit script for s of the same cost but leave $s_{p[1]}$ alone.

Case 1: If $s_{p_j[1]}$ is not part of any well-formed substring.

$s^r_{q_j}$ corresponds to $s_{p_j[1]}$. By performing the same edit operation as the optimal algorithm at $s_{p_j[1]}$ and also at $s^r_{q_j}$. The resultant string s' after the edit at $s_{p_j[1]}$ in s, has edit distance $d - 1$. If well-formed substrings are removed from s', and also from s^r after the edit at $s^r_{q_j}$ to get s'^r, then they both return the same processed string. By the induction hypothesis, s' and s'^r have the same edit distance, $Cost(s') = Costs'^r = (d - 1)$. Since

$Cost(s^r) = Cost(s'^r)+1 = (d-1)+1 = d$, we prove that $Cost(s^r) = Cost(s)$ in this case.

Case 2: $s_{p_j[1]}$ is part of some well-formed substring w.

There are few cases to be consider based on whether $s_{p_j[1]} \in T$ 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 $s_{p_j[1]}$ is not part of any well-formed substring. Then by the same argument as in Case 1, the proof is established.

Subcase 2(a): $s_{p_j[1]} \in T$. By removing well-formed substrings if any, from the prefix $s_1s_2...s_{p_j[1]}$, we must have $s^r_1s^r_2...s^r_{q_j}s_ms_{m+1}...s_{p_j[1]}$ for some integer $m \geq 1$, where all $s^r_1s^r_2...s^r_{q_j}s_ls_{l+1}...s_{p_j[1]}$ are open brackets. The edit at $s_{p_j[1]}$ can be either deletion, or substitution to some other open bracket, or substitution to closed bracket. For the i-th symbol s, let $match[i]$ denote the position of the symbol in s to which it matches.

Deletion: If edit at $s_{p_j[1]}$ is a deletion, $s_{match[p_j[1]]}$ has to be matched with some s_{r_1}, $r_1 < p_j[1]$, otherwise edit costs at $s_{p_j[1]}$ 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 $s^r_1s^r_2...s^r_{q_j}$, we can simply match $s_{p_j[1]}$ with $s_{match[p_j[1]]}$ and delete s_{r_1} instead. This reduces the edit distance by 1 and we can use the same argument as in Case 1. Otherwise, s_{r_1} is part of $s_{m+1}...s_{p_j[1]-1}$. Then the bracket matches s_{r_1} is $s_{match[r_1]}$, which must be matched with some s_{r_2}. Again, if s_{r_2} is part of $s^r_1s^r_2...s^r_{q_j}$, we can instead delete s_{r_2} and match $s_{p_j[1]}$ with $s_{match[p_j[1]]}$ and s_{r_1} with $s_{match[r_1]}$. Otherwise, s_{r_2} is part of $s^r_1s^r_2...s^r_{q_j}$ and we continue in this fashion until we reach some s_{r_h} in $s^r_1s^r_2...s^r_{q_j}$. We then instead can delete $s_{r[h]}$ and match the corresponding pairs in w for $s_{m+1}, ..., s_{p_j[1]}$.

Substitution : If edit at $s_{p_j[1]}$ is a substitution to some different open bracket to match with, say, $s_{r'_1}$ and $r'_1 \neq match[p_j[1]]$, such substitution breaks the well-formedness of w and at least one more edit is required. we will prove in the following why this scenario will not happen. If $s_{r'_1}$ is not part of w, we could just delete $s_{r'_1}$ and match all o f w to save one in edit distance. Otherwise, if $s_{r'_1}$ is part of w , we could have lower the edit distance by 2 without touching w–this contradicts the optimality. Hence, no optimal algorithm considers substituting at $s_{p_j[1]}$ a different open bracket. Similar argument removes the possibility of substituting a closed bracket at $s_{p_j[1]}$.

Subcase 2(b): $s_{p_j[1]} \in \bar{T}$. If we remove well-formed substring from the prefix $s_1 s_2 ... s_{p_j[1]}$, we must have ${}^r_1 s^r_2 ... s^r_{q_j} s_m ... s_{m'} s_{p[1]}$ where $s_{m'} \asymp s_{p_j[1]}$.

Deletion: If edit at $s_{p_j[1]}$ is an deletion, $s_{m'}$ must be matched with some s_{r_1}, $r_1 > p_j[1]$. Instead, it is possible to match $s_{m'}$ and $s_{p_j[1]}$, and delete s_{r_1}. This defers the first edit without affecting the optimality giving a contradiction.

Substitution : If edit at $s_{p_j[1]}$ is a substitution to a different closed bracket, which could never happen. Because such operation requires editing $s_{m'}$ as well. If edit at $s_{p_j[1]}$ is a substitution to a new open to match some $s_{r'_1}$, $r'_1 > p[1]$, we can delete $s_{r'_1}$ and match $s_{l'}$ to $s_{p[1]}$ instead, contradicting the fact that the considered optimal algorithm defers the first edit as much as possible. □

Efficient in most cases, however, there are still 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. As the average of an document tree should not be too deep(less than 7 on average.), and errors are not that frequent comparing with string length, well-formed substrings are not rare. We investigate its effectiveness on real data in Section 3.5.2 in detail.

3.4 An Incremental Approach based on BAB

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.

3.4.1 Branching-and-Bounding Algorithm

Unlike the dynamic programming algorithm which computes every possible matching between $s[i]$ and $s[j]$, the Branching-and-Bound(BAB) algorithm s-martly decides where to stop to start a checking, motivated by Theorem 3.8.

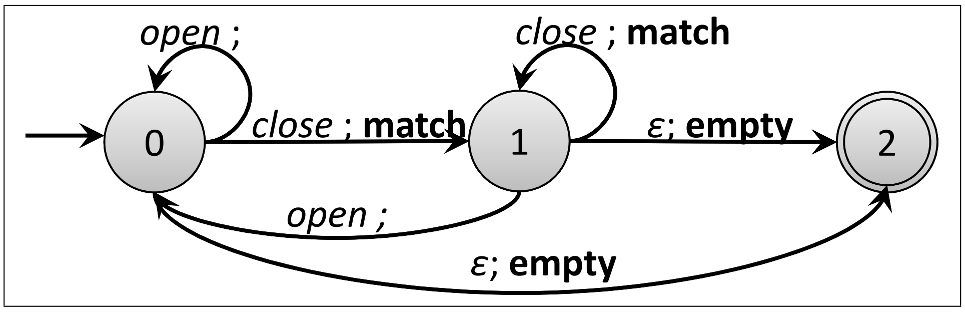


Figure 3.1: Automata for Grammar G_T

Figure 3.1 shows how a well-formed string satisfying G_T is accepted by the automata. This automata simply has three states: 0, processed an open bracket; 1, processed one matching close bracket; and 2, the sink state. If the string finally stops at state 2, it is accepted by the grammar. During string processing, we use a stack to keep all unmatched open brackets, and read one bracket at a time from the string. So each edge is associated with a state transition condition $\langle bracketType; matched \rangle$. For example, from state 0 to 1, the condition is ⟨*close* ;**match**⟩, which means if the bracket under processing is in $\bar{T}$ and it **matches** with the open bracket at the top of the bracket. A special condition is $\langle \varepsilon,$ **empty**⟩, which means all brackets have been processed and the stack is also empty. Any deviation from the automata is a sign of erroneous input. Therefor, we only need to stop when we see such a sign, rather than suspecting every pairs of brackets.

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. As the string is parsed naturally from left to right, one bracket at a time, for simplicity, we use "left" to denote the top open bracket on the stack, and "right" the next bracket from the input string.

Given a mismatch of types between an adjacent open-close pair, there are only five edit operations that need to be considered.

(a) Delete the open bracket on the left;

(b) Delete the close bracket on the right;

(c) Substitute the left or right bracket to make a matching pair;

(d) Substitute the open bracket on the left to a close bracket;

(e) 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 theorem establishes the correctness of our algorithm.

Theorem 3.9 By considering edit operations at only one of the following scenarios, a sequence of choices exist that leads to the optimal edit distance: (a) an empty stack when a close bracket occurs; (b) a non-empty stack when the string has terminated; and (c) 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. As proved previously in Theorem 3.8, removal of well-formed substrings preserves the minimum edit distance. For simplicity, given a string s with d edit distance, we focus on its counterpart s' with well-formed substrings removed.

If s is well-formed, i.e., $d = 0$, it is obvious that the algorithm returns the optimal answer 0, as the stack is empty in the end and never stops for branching.

If s is non-well-formed, i.e., $d > 0$, the string s' after well-formed substring removal, also has an edit distance of d and contains no well-formed substrings at all. Then s' must fall into one of following three cases: a) s' is a run of open brackets, b) s' starts with a close bracket, i.e., $s'_1 \in \bar{T}$. c) s' has a sequence of open brackets followed by some close bracket that does not match the bracket prior to it. So, to repair the string s', any optimal algorithm must make some edit operations in all these scenarios. In case a), any canonical repair by paring up the open brackets and deleting at most one open bracket (if there is odd number of brackets) gives the optimal solution. In case b), the edit operations are either deleting the leading close bracket or substituting it with some open bracket. In case c), the choices are more complex and we will explain one by one. There are in total five alternative edit operations here:

(i) delete the open bracket from the stack

(ii) delete the close bracket from the input string

(iii) substitute the left(open) or right(close) bracket to make a matching pair

(iv) substitute the left(open) bracket to match the top-second tag from the stack, if possible

(v) substitute the right(close) bracket to a *ghost* open bracket. A *Ghost* open is an open bracket whose type is not fixed and can be matched with any close bracket.

If some optimal algorithm $\mathcal{A}$ chooses any of the above edit operations except (v) to repair s, and results in a string with edit distance $(d-1)$, then there is a branch that considers exactly the same edit followed by well-formed substring removal, leading to a string with edit distance $(d-1)$. But if the optimal algorithm $\mathcal{A}$ selects the operation (v) in case c) by substituting the close bracket to some $x \in T$ to repair s. In our branching algorithm, we offer an option to replace the close bracket with a *ghost* open bracket. As the *ghost* open can be resolved to any open bracket at that point, and hence to x as well. There for, the resultant string has an edit distance of $d-1$ 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. □

3.4.2 Greedy Heuristics

By visiting all possible branches at each erroneous point leads to high complex when the number of errors is large. To speed up the algorithm with certain loss in accuracy, 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**: Pre-compute 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 pre-computed and stored globally. When $|T| = 1$, MinCost also finds an optimal cost repair.

Unfortunately, both of these heuristics may 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.

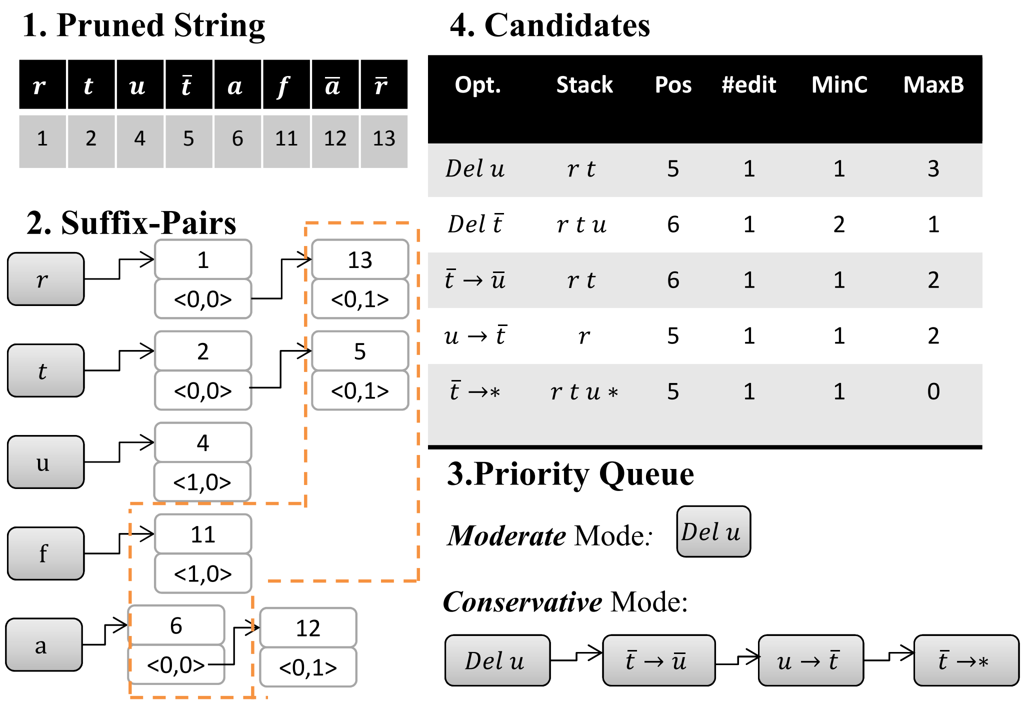
- **Conservative:** Try all the choices in the union that Max-Benefit or Min-Cost gives.
- **Moderate:** Try all the choices in the multi-set intersection of the choices that Max-Benefit or Min-Cost gives; if the intersection is empty, then try all the choices from their union.
- **Liberal:** Select one choice at random from the multi-set 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 3.5.

3.4.3 Implementation for Branching Strategies

Figure 3.2 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 1.1(a). Two global structures, bracket list and suffix pairs, are preprocessed in a single scan to assist the procedure. The bracket list contains the tokenized brackets and their index positions after well-formed substring pruning.

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



1. Pruned String
r t u $\bar{t}$ a f $\bar{a}$ $\bar{r}$
1 2 4 5 6 11 12 13
2. Suffix-Pairs
r 1 <0,0> 13 <0,1>
t 2 <0,0> 5 <0,1>
u 4 <1,0>
f 11 <1,0>
a 6 <0,0> 12 <0,1>
4. Candidates
3.Priority Queue
Moderate Mode: $Del\ u$
Conservative Mode: $Del\ u$ $\bar{t} \to \bar{u}$ $u \to \bar{t}$ $\bar{t} \to *$

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

Figure 3.2: Illustration of Branch-and-Bound Algorithm

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 3.2. 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 be 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.

3.5 Experimental Evaluation

This section gives a thorough experimental evaluation of methods for the Bracket Language Edit Distance Problem, against the *Dynamic Programming* (DP) and *branch-and-bound* algorithms. Table 3.1 lists the detailed description of modes, used in the *branch-and-bound* algorithm, where S_{all} considers the whole choice set, S_b is the set chosen by the **MaxBenefit** heuristic and S_c chosen by the **MinCost** heuristic, $\mathcal{R}()$ is the function that selects one choice from the set randomly.

3.5.1 Experiments Setup

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

Table 3.1: Summary of Methods

Mode Name	Choice Set	Multi-Repair
Exhaustive	S_{all}	Yes
Conservative	$S_b \bigcup S_c$	Yes
Moderate	If $\lvert S_b \bigcap S_c \rvert > 0$, $S_b \bigcap S_c$ Else $S_b \bigcup S_c$	Yes
Liberal	If $\lvert S_b \bigcap S_c \rvert > 0$, $\mathcal{R}(S_b \bigcap S_c)$ Else $\mathcal{R}(S_b) \bigcup \mathcal{R}(S_c)$	Yes
MinCost++	S_c	Yes
MaxBenefit++	S_b	Yes
MinCost	$\mathcal{R}(S_c)$	No
MaxBenefit	$\mathcal{R}(S_b)$	No
Hybrid	$\mathcal{R}(S_c \bigcup S_b)$	No
Random	$\mathcal{R}(S_{all})$	No
Rule-based	N/A	No

Centos 5.8. We used the following two real data sets:

- *BGP*[2] real-time routing information provided by BGPmon.We used a portion of the stream output over some time interval.
- *Tree Bank*[3] annotated linguistic text, with average depth 7.8 and max depth 36. We extracted random subtrees with max-depth no less than 20 and merged them together.

Both data sets normally satisfy the grammar G_T and only the tag subsequences are retained.

Error Model and Parameter Setting: We choose from among six different operations with equal probability to inject errors into a given well-formed string,

[2] http://bgpmon.netsec.colostate.edu/
[3] http://www.cis.upenn.edu/~treebank/

as listed in Table 3.2. The detail settings of string length and error number are given in Table 3.3.

Table 3.2: 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 $(T \cup \bar{T})$

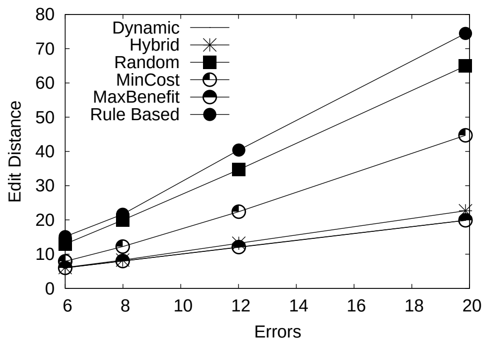
Metrics: Each experiment was repeated 100 times, and we report *Average Running Time* and *Average Edit Distance*.

3.5.2 Single Repairs

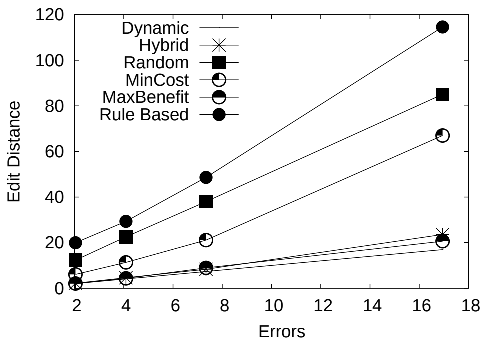
Table 3.3: Data Set Properties

Parameter	**Range**
String Length(TreeBank)($\times 10^3$)	$1, 2, \mathbf{4}, 8$
String Length(BGP)($\times 10^3$)	$10, 20, \mathbf{40}, 80$
Error Number(TreeBank)	$2, 4, \mathbf{8}, 16$
Error Number(BGP)	$6, 8, \mathbf{12}, 20$
Well-formed Substring Eliminate	**Yes**, No
top-k threshold k	$1, \mathbf{5}, 10, 15, 20$

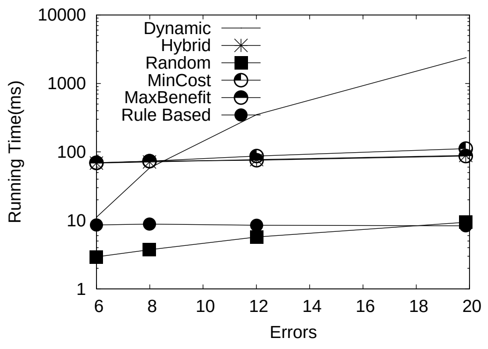
We compared methods that are designed to return a single repair, including DP as well as MaxBenefit, MinCost and Hybrid, where Hybrid randomly picks one choice either given by MinCost or by MaxBenefit. As a baseline, we also tried three rule-based heuristics for handling open-close mismatches: one which 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.



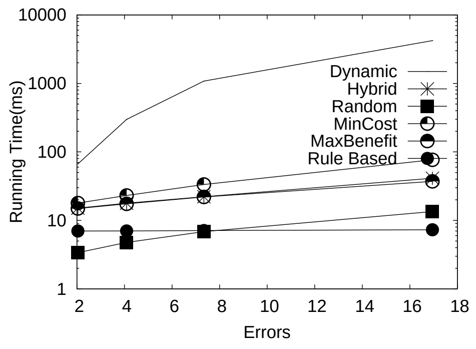
(a) On *BGP* data



(b) On *Tree Bank* data



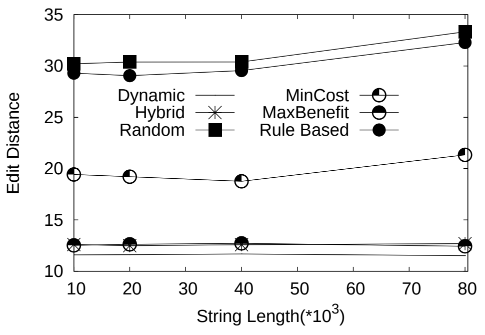
(c) On *BGP* data



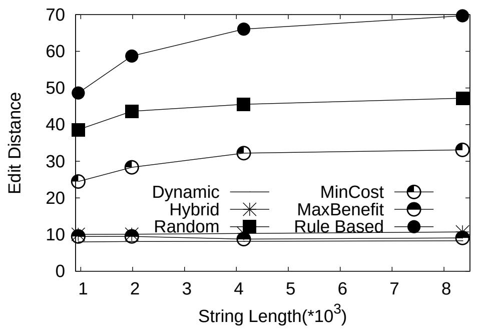
(d) On *Tree Bank* data

Figure 3.3: Single Repair, *Error Number*

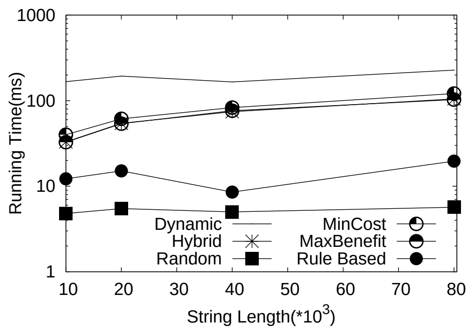
Figure 3.3 shows the edit distance and running time as a function of the number of errors. For *BGP*, the errors ranged from 6 to $F20$ with initial string length fixed at $40,000$; after well-formed substring pruning, the string length was significantly reduced to the range $[20,160]$. For *Tree Bank*, the error



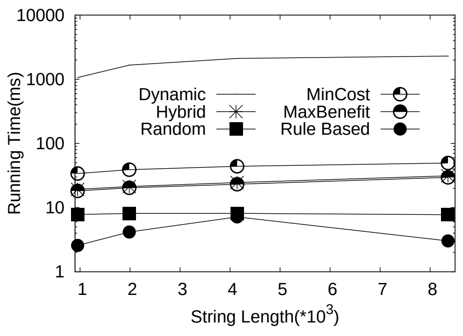
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



(d) On *Tree Bank* data

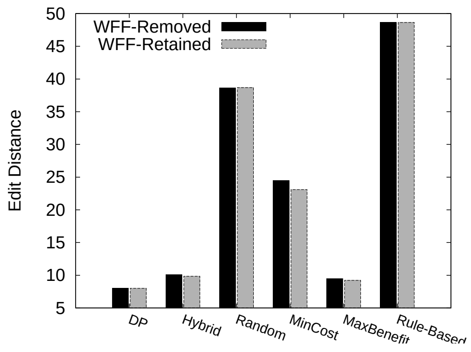
Figure 3.4: Single-Repair, *String Length*

ranged from 2 to16 with string length fixed at $4,000$; after pruning the length was reduced to $[40, 280]$.

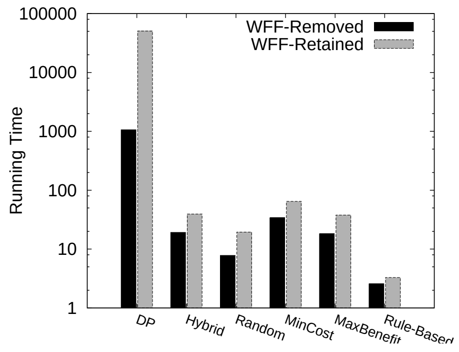
While DP gave the smallest edit distance (it is optimal), it was also the slowest in almost all cases. The running time of DP increased significantly with the number of errors, even with well-formed substring pruning: the string length after pruning increased from 20 to 160 on *BGP* and from 40 to 280 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 3.4 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 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.

3.5.2.1 Retaining Well-formed Substrings



(a) Edit Distance



(b) Running Time

Figure 3.5: Well-formed Substring

As shown in Section 3.3.2, well-formed substring pruning, does not affect the edit distance of DP. However, it significantly improve the 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 3.5 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.

3.5.3 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. When $K = 1$, we prune off as many branches as possible, while $K > 1$, we preserve as many repairs, so the prefix-string hashing function is adopted, to retain nodes with the same stack and string but different repair prefixes.

Figure 3.6 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 Figure 3.6(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 3.7 shows performance versus string length ranging from $10,000$ to

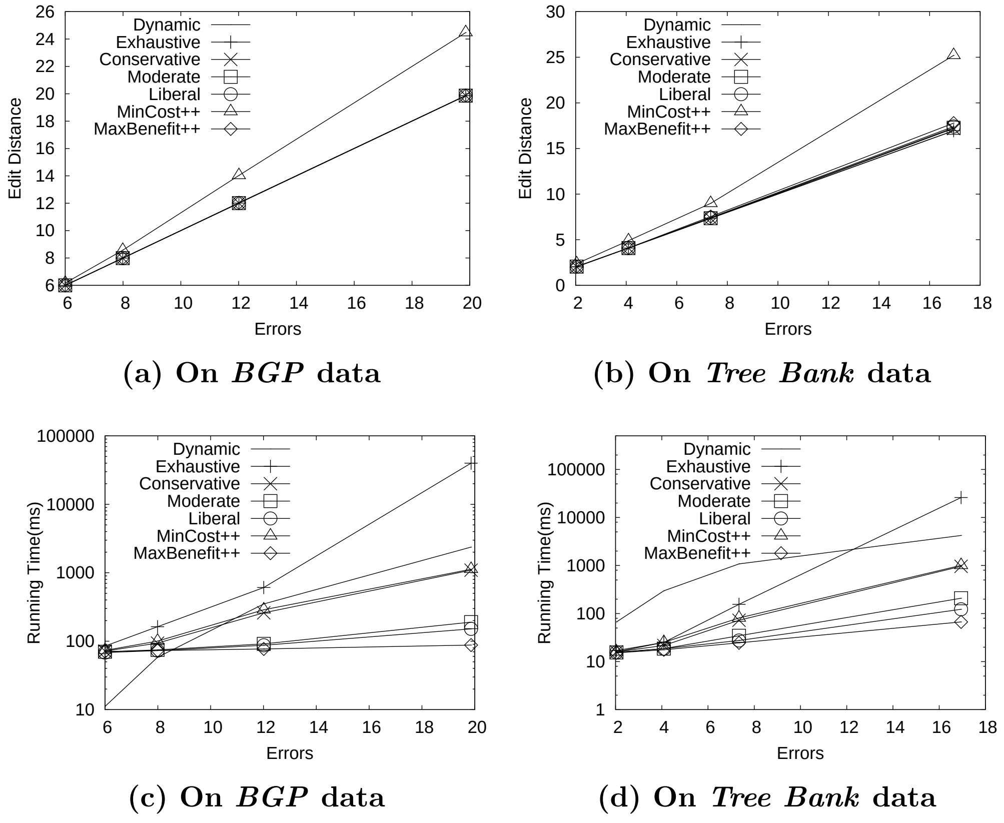
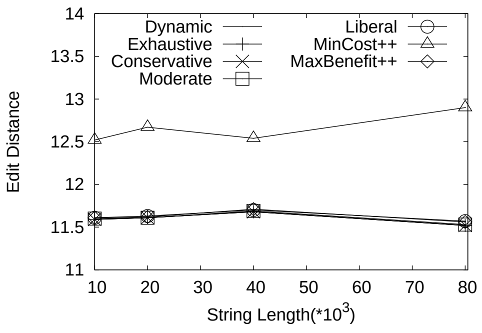


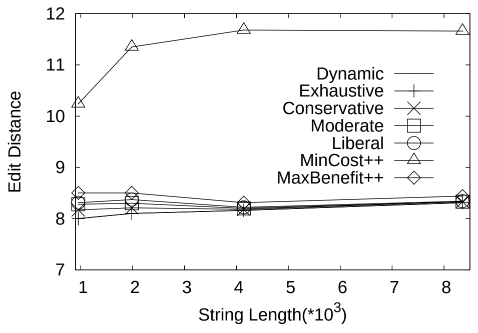
Figure 3.6: Multi-Repairs, *Error Number*

$80,000$ (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.

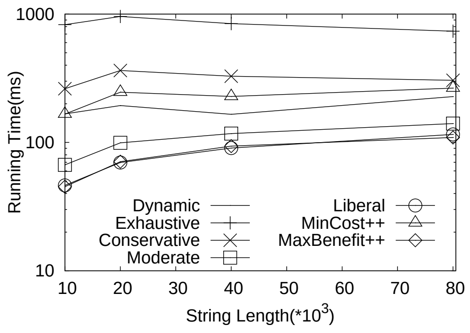
Multi-Repairs Performance: Figure 3.8 illustrates the performance for finding 5 repairs when the number of errors increases from 6 to 20 on *BGP* with string length fixed at 40,000) and from 2 to 16 on *Tree Bank* (with string length fixed at 4,000). When error number is 20 on *BGP* dataset, it takes Exhaustive an



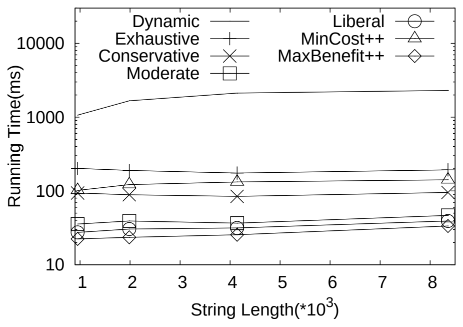
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



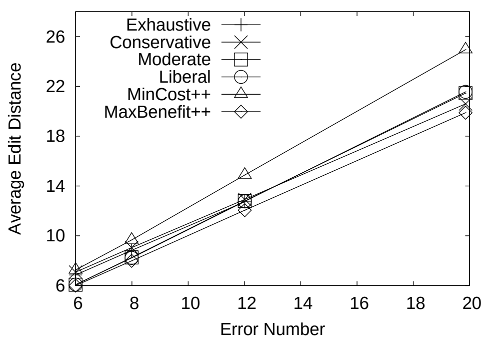
(d) On *Tree Bank* data

Figure 3.7: Multi-Repair, *String Length*

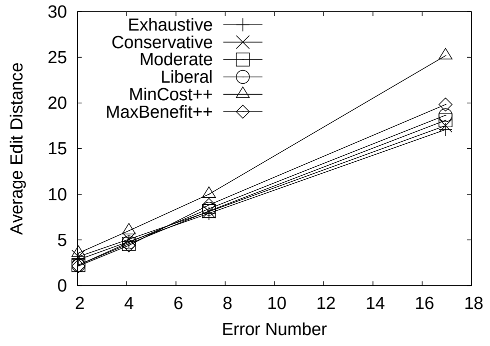
extremely large amount of time, so we do not plot the results there.

Figure 3.9 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 the well-formed substring removed, the string length decreases significantly to 130 on *Tree Bank*, and 60 on *BGP* dataset.

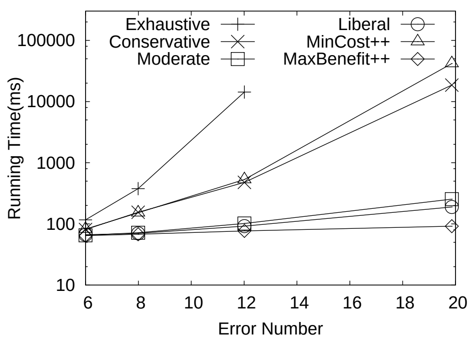
Exhaustive, Conservative and MinCost++ are 10 times slower than Moderate, Liberal and MaxBenefit++ since the former methods return more repairs than the latter do. MinCost++ is less accurate than Exhaustive and Conservative, but is faster on *BGP*; on *Tree Bank*MinCost++ is less accurate and comparable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



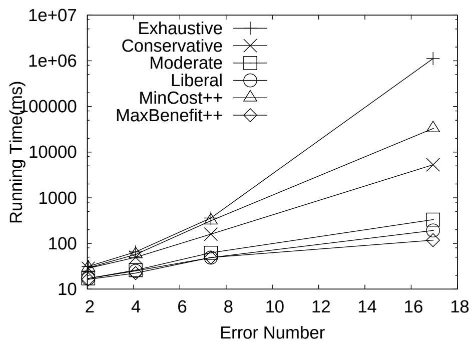
(a) On *BGP* data



(b) On *Tree Bank* data



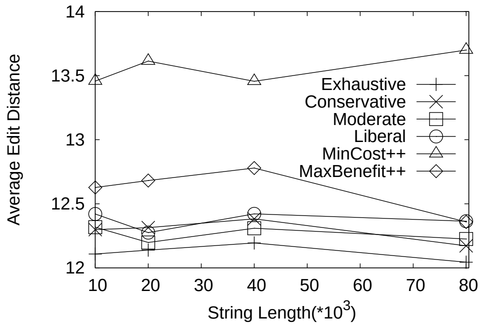
(c) On *BGP* data



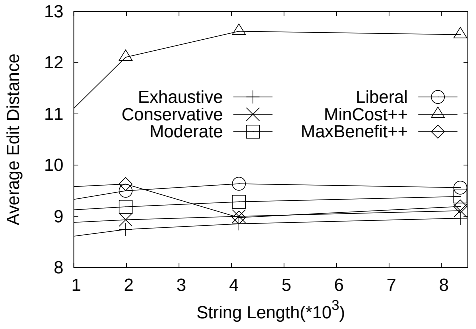
(d) On *Tree Bank* data

Figure 3.8: *top-5* Repairs, *Error Number*

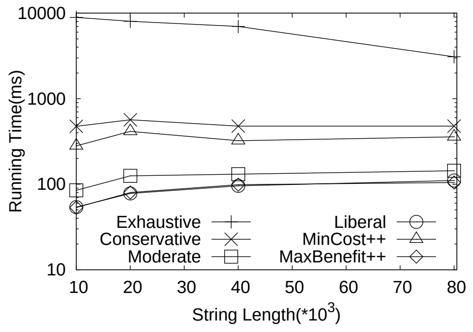
K-**Repairs Performance:** We evaluated the performance of finding up to K repairs, for $K \in \{1, 5, 10, 15, 20\}$, with string length 40,000 for *BGP* and 4,000 for *Tree Bank*. Note that not all methods were able to obtain K repairs. Moderate, Liberal and MaxBenefit++ run faster by aggressively pruning; therefore, they result in fewer total repairs. Figure 3.10 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



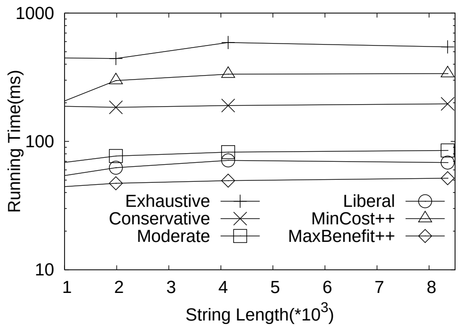
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



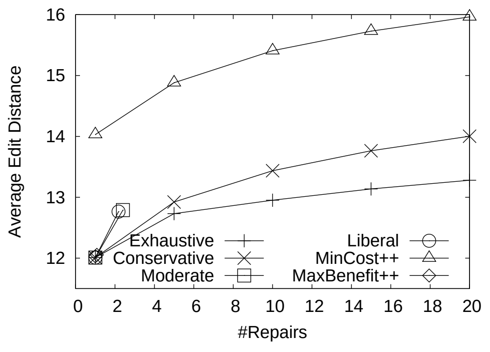
(d) On *Tree Bank* data

Figure 3.9: *top-5* Repairs, *String Length*

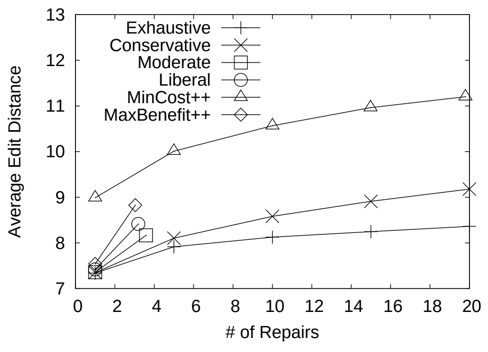
repairs, the running time grows linearly in K.

3.6 Conclusions

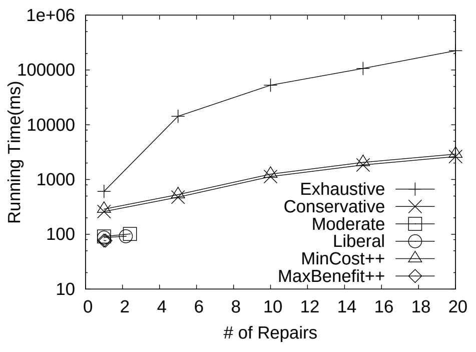
In this chapter we investigate the problem on repairing semi-structured document when open- and close- tags are unmatched. Two algorithms were proposed to meet various user demands, and user can trade off accuracy for efficiency by combing the heuristics in different ways. The Branch-and-Bound algorithm, apart from returning multiple canonical repairs, has another advantage: it works especially good for streaming data, such as log file, where the size is not given ahead and user can choose to terminate the algorithm any time to get a list of



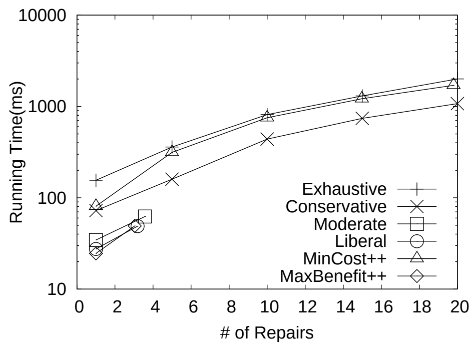
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



(d) On *Tree Bank* data

Figure 3.10: *top-k*, Scalability

repairs for the documents listened so far.

Besides tags (or brackets), there are many other components, e.g., text values, attributes, IDREF (cross-reference) in the documents. Even when the grammar is unknown, these components are good clues and can be leveraged for a more judicious repair.

Chapter 4

REPAIR WITH RESTRICTED TEXT OCCURRENCE

A commonly occurring pattern for semi-structured documents, especially those used for data interchange and storage, is that text content must occur and only occur between a matching pair, which the norm of files like JSON, as well as XML files encoding JSON. In this chapter, we study how to exploit this pattern to compute a more judicious repair. We also show in the experimental evaluation to which extent the injected errors can be undone by comparing the repaired documents with the original documents, measured by tree edit distance.

4.1 Motivation

Tags are bricks in constructing semi-structured documents, and open- and close-tags matching is the universal rule that every such document must obey. Even as flexible as HTML, which allows the input to be non-wellformed, it is the HTML parser that takes over the burden of making missing tags up. Apart from the tag-matching rule, we observe another important constraint many documents inherent: text value must be embedded within pairs of tags, e.g., text values are usually leaf nodes in an XML DOM tree, rather than appearing arbitrary under any tags, or being sibling of some other tags. Such constraint is rather common and can be found in many types of semi-structured documents, especially those for data interchange and data storage, for instance JSON files, RSS feeds, system config files. 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) [29].

There has been many work on approximate matching of trees which has been applied to finding semantically relevant XML documents [30, 50, 80]. Unfortunately none of these works apply to such setting since the input is not even well-formed in the sense of tags and cannot be parsed or represented as a tree. Hence, we study in this chapter how to exploit the text constraint to aid in finding a repair, which would be more meaningful.

4.2 Problem Definition

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 4.1 Well-formed String

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 4.2 Well-formed Bracketed Language with Text

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 4.1 Let $W = \{w\}$, and $T = \{a, b\}$, and $\bar{T} = \{\bar{a}, \bar{b}\}$. 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.

We define the edit distance $E(s, s')$ between two strings, s and s', in $(W \cup T \cup \bar{T})^*$ as the minimum number of *bracket* insertions, deletions and substitutions needed to transform s into s'; only the brackets and not the words in the strings are considered for edit operations.

Definition 4.3 Bracketed Language with Text Edit Distance Problem

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 4.2 Given then string $s = ab\bar{b}w\bar{a}$. When there is no constraint on text, the word w is ignored and the string is well-formed. But when text must exist as leaf node, it has edit distance 2, and can be repaired to $abw\bar{b}\bar{a}$ using one deletion and one insertion, or $aw\bar{a}$ with two deletions.

4.3 An Optimal Solution using Dynamic Programming

As shown in the example in Chapter 1 Figure 1.2, if we consider only the bracket subsequence of the string and apply the algorithm proposed in Theorem 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 matching lower-bounds that for this case). Therefore, we need to design a new algorithm.

Algorithm 4.1 presents the pseudocode for the dynamic programming. The algorithm runs in $O(n^3)$ time and requires $O(n^2)$ space, where n is the number of brackets. Given some substring $s_i...s_j$, the algorithm first checks if it contains some word (that is, $s_k \in W$ for some $k \in [i,j]$) and, if not, deletes $s_i...s_j$, resulting in cost $C[i,j] = j-i+1$. Otherwise, either s_i and s_j could be edited to match brackets surrounding a well-formed substring (the $S \rightarrow xS\bar{x}$ 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 $S \rightarrow xw^+\bar{x}$ production); or else the string could be broken into two adjacent well-formed substrings (the $S \rightarrow SS$ production). 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 $s_is_{i+1}...s_j$ contains a word, the recurrence is:

Algorithm 4.1: Dynamic Programming for Tag-with-Text

```
Input: tokenized string $s = s_1 ... s_n$
Output: edit distance $\min_{s'} E(s, s')$ where $s'$ is well-formed
1  forall the $i$ from 0 to $n-1$ do
2      forall the $j$ from $i+1$ to $n$ do
3          if $s_i \asymp s_j$ then
4              $B[i,j] \leftarrow 0$
5          else
6              if $s_i \in T$ or $s_j \in \bar{T}$ then
7                  $B[i,j] \leftarrow 1$
8              else
9                  $B[i,j] \leftarrow 2$
10 forall the $i$ from 1 to $n$ do
11     if $s_i \in W$ then
12         $B[i,i] \leftarrow 2$
13     else
14         $B[i,i] \leftarrow 1$
15     forall the $\ell$ from 1 to $n-1$ do
16         forall the $i$ from 1 to $n-\ell$ do
17             if $s_k \notin W, \forall k \in [i,j]$ then
18                 $C[i,j] \leftarrow j-i+1$
19             else
20                 $j \leftarrow i+\ell$
21                 $C[i,j] \leftarrow B[i,j] + D[j-1,i+1]$
22                 $C[i,j] \leftarrow \min(C[i,j], B[i,j] + C[i+1,j-1])$
23                 forall the $k$ from $i$ to $j-1$ do
24                     $C[i,j] \leftarrow \min(C[i,j], C[i,k] + C[k+1,j])$
25 return $C[1,n]$
```

$$C[i,j] = min \begin{cases} B[i,j] + C[i+1,j-1], & i < j \\ B[i,j] + D[i+1,j-1], & i < j \\ \min_{i \le k \le j-1} C[i,k] + C[k+1,j], & i < j \end{cases} \tag{4.1}$$

, where $B[i,j]$ be the smallest cost of editing s_i and s_j to match, defined as:

$$B[i,j] = \begin{cases} 0, & match(s_i, s_j) \\ 1, & (s_i \in T \wedge s_j \in W) \vee (s_i \in W \wedge s_j \in \bar{T}) \\ 2, & (s_i \in W \wedge s_j \in W) \vee (s_i \in W \wedge s_j \in T) \vee (s_i \in \bar{T} \wedge s_j \in W) \end{cases} \tag{4.2}$$

Theorem 4.4 Algorithm 4.1 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, i.e., s_i. If $s_i \in W$, two matching open and close parenthesis need to be inserted to surround s_i, resulting in edit distance 2, since words cannot be edited. If $s_i \in T$, the minimum edit distance is 1, which can be achieved by deleting s_i (by rule $S \rightarrow \varepsilon$), as we do not allow brackets without any words.

For substrings of length 2, $s_i s_{i+1}$, $B[i,i+1]$ computes the edit distance correctly. As $C[i,i+1] = B[i,i+1]$ in this case, we claim the result of $C[i,i+1]$ is correct. These serve as the base cases. The detail of $B[i,j]$ can refer to Equation 4.2

Suppose, by induction hypothesis, Algorithm 4.1 correctly computes minimum edit distance for all substrings of length at most l. We now take any substring of length $l+1$. Without loss of generality, let it be $s_m s_{m+1} .... s_{m+l+1}$. Taking m for instance as the position where first edit occurs by some optimal edit script P, such an optimal algorithm $\mathcal{A}$ has a handle of options as following and we prove by showing that Algorithm 4.1 considers all the cases.

- *Delete s_m*

  In this case, $s_m \in T \cup \bar{T}$, and the minimum edit distance is $1+C[m+1, m+l+1]$. In Algorithm 4.1,it has the choice to set edit distance as $C[m, m+l+1] = C[m, m] + C[m+1, m+l+1]$. Based on the induction hypothesis, $C[m, m$ and $C[m+1, m+l_1]$ have been correctly computed, so the answer of $C[m, m+l+1]$ is also correct in this case.

- *Matches s_m to some s_j, $j > m$, and $s_m, s_j \in T \cup \bar{T}$.*

  First we assume neither s_m nor s_j is a word.

  (a) $j = m+l+1$ and $s_m \asymp s_j$. In this case, the minimum edit distance is $C[m, m+l+1] = \min(C[m+1, m+l], D[m+1, m+])$. The algorithm in this case computes $B[m, m+l+1] = 0$ and has the choice to set edit distance as $C[m, m+l+1] = \min(B[m, m+l+1] + C[m+1, m+l], B[m, m+l+1]+D[m+1, m+l])$; by induction hypothesis, the second term in RHS(Right Hand Side) is computed correctly.

  (b) $j = m+l+1$, $s_m \in T \cup \bar{T}$ and $s_j \in \bar{T}$, and s_m is substituted to match s_j. In this case, the minimum edit distance is $C[m, m+l+1] = \min(m + C[m+1, m+l], 1 + D[m+1, m+l])$. The algorithm in this case correctly computes $B[m, m+l+1] = 1$ and has the choice to set edit distance as $C[m, m+l+1] = \min(B[m, m+l+1] + C[m+1, m+l], B[m, m+l+1]+ D[m+$

$1, m+l]$); by induction hypothesis, we get $C[m+1, m+1]$ correctly, so the answer to $C[1, m+l+1]$ is also correct.

(c) $j = m+l+1$, $s_m \in \bar{T}$ and $s_j \in T$. To form a valid match, the minimum edit distance is $C[m, m+l+1] = \min(2 + C[m+1, m+l], 2 + D[m+1, m+l]$. The algorithm in this case correctly computes $B[m, m+l+1] = 2$ and has the choice to set edit distance as $C[m, m+l+1] = \min(B[m, m+l+1] + C[m+1, m+l], B[m, m+l+1] + D[m+1, m+l])$; by induction hypothesis, the second term in RHS is computed correctly.

(d) $m < j < m+l+1$. In this case, the substring is regarded as consisting of several adjacent substrings, and the minimum edit distance is $C[m, m+l+1] = C[m, j] + C[j+1, m+l+1]$. This alternative is also covered by Algorithm 4.1 as show in Equation 4.2. As both $C[m, j]$ and $C[j+1, m+l+1]$ have correct result, $C[m, m+l+1]$ is computed correctly by induction hypothesis.

- *Matches s_m with s_j, and $s_m \in W$ or $s_j \in W$*

  In this case, we must insert some open bracket or possibly a corresponding close bracket as well, depending on which of the two is a word, or both are. The success of the algorithm is highly relies on the correctness of Equation 4.2.

  If $s_m \in W$ and $s_j \in \bar{T}$, the best edit is achieving by inserting a matching open bracket before s_m at a cost of 1.When $s_j \in W$ and $s_i \in T$, the repair is achieved by inserting a matching close bracket after s_j with one edit. These two cases are correctly covered in Equation 4.2

case(1).

If $s_m \in W$ and $s_j \in T$, we have to perform at least two edits: substituting s_j to some close bracket and inserting a matching open bracket in front of s_m. What's more if $s_j \in W$ and $s_m \in \bar{T}$, the cost is also 2, by substituting s_m to some open bracket and inserting a matching close bracket after s_j. $B[m, j]$ is also equal to 2 according to case(3) in Equation 4.2.

Finally if both s_m and s_j are both words, none of them could be deleted or substituted, but are repaired by inserting matching open and close brackets before and after s_m and s_j respectively with two edits. Equation 4.2 covers this alternative in case (3).

These options are exhaustive, as Algorithm 4.1 considers all the options, computing edit distance correctly in all these cases and returning the minimum. Hence, Algorithm 4.1 computes minimum edit distance for any substring of length $l + 1$ correctly. Therefore, by induction, the proof is established. $\square$

4.3.1 Well-formed Substring Removal

As discussion in Chapter 3, well-formed substrings removal could significantly speed up the dynamic programming algorithm while preserving the optimality of edit distance. We can preprocess the string to enable faster computation by removing well-formed substrings using a stack.

The exact value of each text is not our concern but the position where each of them appear is. So the input string can be viewed as a list of *String Blocks*, tokenized by text.

Definition 4.5 String Block

A *string block*, denoted B_i, is a substring of brackets that appears between two consecutive occurrence of texts t_k and t_{k+1}, or before the first text, or after the last text.

For example, the string $s = aaaw\bar{a}\bar{a}aw\bar{a}\bar{a}\bar{a}\bar{a}$ has three blocks, $B_0 = aaa, B_1 = \bar{a}\bar{a}a, B_2 = \bar{a}\bar{a}\bar{a}\bar{a}$. According to Definition 4.1, only brackets that belong to different blocks can be matched.

Unfortunately, such removal 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. Luckily the result will not be arbitrary bad but stays within at least a factor of 2 after well-formed substring pruning, which is given in Theorem 4.6, and during the experiments we find the difference on real data is much better than 2-approximate in most cases.

Suppose we have a new edit distance function which is exactly similar to the original edit distance, except that two consecutive open brackets (resp. close brackets) in a block can be deleted at a cost of 1. Obviously, if this new edit distance function has minimum distance d, the optimal edit distance for the original problem is bounded by $2d$ (we pay 2 unit cost to delete two open (resp. close) brackets in a block). The main insight is to consider this new function, and prove if the optimality of new edit distance function is preserved with well-formed substring elimination, and hence guarantees 2-approximation to the original problem.

Theorem 4.6 Removing well-formed substrings obtains a 2-approximation on the new edit distance.

Proof. Let s be the original string and s^r be the string obtained after removing well-formed substrings from s. Let decompose s and write it as concatenation of substrings $r_0 s_1 r_1 s_2 r_2 ... s_k r_k$, where $s^r = s_1 s_2 ... s_k$ and each of r_i, $i = 0, 2, .., k$ are well-formed substrings, with the possibility of r_0 and r_k being empty.

First within each block, consider the substrings of the form $T^* \bar{T}^*$(a block is a mixture of $(T \cup \bar{T})^*$), though they are well-formed according to the grammar *gramparen*, these substrings do not comply to grammar $G_{T,W}$. Therefor such substrings will be kept and are part of s^r as well. Perform the same set of edits done by some optimal algorithm $\mathcal{A}$ on both s and s^r, and let the resultant strings be t and t' respectively. 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 surrounded by at least one open and close bracket. The resultant strings after these edits are denoted as 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 brackets in z (z') and consider an optimal algorithm for the Bracket Language Edit Distance Problem, 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 edit distance of z (z') given by the Bracket Language Edit Distance Problem, or Tag-Only Edit distance for short. 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 Theorem 3.8, 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. □

4.4 Incremental Approach Based on BAB

The Algorithm 4.1 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 matching case. We start by giving pushdown automata in Figure 4.1 that allows a well-formed string to be verified.

There are five states in the automata, with directed edged labeled with transition condition. State 0 and State 4 are the start and sink states. The other three states represent the status during processing a string: with an open bracket x, or open bracket with word xw, or open bracket with word with close bracket $(xw\bar{x})^*$. We have shown in Chapter 3 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 in this problem: only scenarios not covered by the automaton in Figure 4.1 are considered. We show how to deal with these scenarios in Table 4.1.

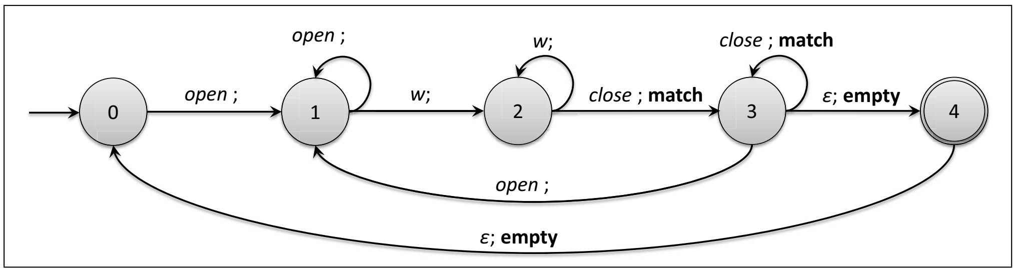


Figure 4.1: Automata for Grammar $G_{T,W}$

By contrast, there are nine scenarios based on the automaton in Figure 4.1. 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:

- 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.

Table 4.1: State Transition Table

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

- 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.
- If the next token in the string is a word, then delete the close bracket; 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 matching case:

(1) a close bracket occurs immediately after an open bracket;

(2) an open bracket occurs immediately after a word;

(3) a word occurs immediately after a close bracket;

(4) a word occurs as the first token; and

(5) the string terminates in a word.

For these additional scenarios, there are various edit alternatives, which are listed in Table 4.1.

The so-called "clean-up" phase referred to in States 1 and 3 of Table 4.1 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.

Example 4.3 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 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.

Theorem 4.7 The automaton given in Table 4.1 with branches to all the above edit alternatives, obtains a 2-approximation on edit distance.

Proof. The main idea is to show the states considered in our automaton are the only scenarios where repair has to be made if well-formed substrings are removed greedily. Consider an optimal algorithm for the new edit function defined in Theorem 4.6, it can be shown that at each error state, repair choices considered are exhaustive and hence 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 matching case, since it provides a lower bound; the alternatives are sorted with respect to these costs.

4.5 Experimental Evaluation

This section gives a thorough experimental evaluation of methods for Bracket Language with Text Edit Distance Problem, against the *Dynamic Programming* (DP) and *branch-and-bound* algorithms.

4.5.1 Experiments Setup

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 three real data sets:

- *BGP*[1] real-time routing information provided by BGPmon.We used a portion of the stream output over some time interval.
- *Tree Bank*[2] annotated linguistic text, with average depth 7.8 and max depth 36. We extracted random subtrees with max-depth no less than 20 and merged them together.
- *Web Repository*[3] an Web repository of XML documents containing $180,000$ files, 10% of which are reported with begin and end tag mismatched.

[1]http://bgpmon.netsec.colostate.edu/
[2]http://www.cis.upenn.edu/~treebank/
[3]http://data.politicalmashup.nl/xmlweb/

Both *Tree Bank* and *BGP* normally satisfy the grammar $G_{T,W}$. For the XML Collections, we pick a set of 80 RSS feeds which also satisfy the grammar $G_{T,W}$. We use the same error model as in Chapter 3 by choosing from six different operations independently each time and inject it into the documents, and the detail of the error injection model will not be presented.

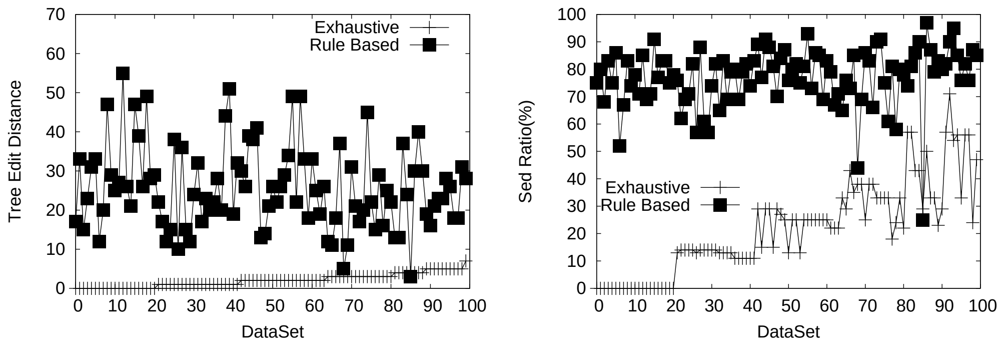
4.5.2 Effectiveness of Edit Distance Approach

To evaluate the goodness of our edit distance based approach, we started from a well-formed string s, injected errors randomly to obtain string s', ran our methods to find a repair s'', and compared s'' against the original well-formed string s. Since the original string and the repair are both well-formed, we are now able to use approximate tree matching algorithms to evaluate the goodness of our repairs. For this, we used the recently developed RTED algorithm from [68], which uses the standard tree edit operations [86]: delete a node and connect its children to its parent, maintaining the order; insert a new node between an existing node, v, and a consecutive subsequence of v's children; and rename the label of a node. We compared our approach against the rule-based heuristics for dealing with open-close tag mismatches: 1) delete the open tag; 2) delete the close tag; and 3) substitute the close tag to match. In the following experiments, we present the best of these three heuristics.

We used a data set of 100 strings from *Tree Bank* consisting of 1,000 tags and around 8 errors, with uniformly distributed errors.

Figure 4.2(a) shows the tree edit distance, where the x-axis indexes all of the 100 data sets sorted by the tree edit distance of Exhaustive repair, which is the branch-and-bound algorithm that tries all five choices at each branch point. There were 20 strings for which our method obtained $d_{TED}(s, s'') = 0$, a complete reversion of the string compared to the original, and over 60% of the strings had $d_{TED}(s, s'') \leq 3$. In contrast, the rule-based heuristics obtained strings with average tree edit distance 25.

In addition, we also did the following comparison. Starting from the original (well-formed) input string s, we injected errors to obtain string s' and computed the *string* edit distance between s and s'. Then we repaired the string to obtain string s'' and computed the string edit distance between s' and s''. Finally we calculated $\frac{d_{SED}(s,s'')}{d_{SED}(s,s')+d_{SED}(s',s'')}$ as a measure of how well the error was "undone", where 0 means exact reconstruction and 1 means the repair resulted in no improvement. We compared our approach with the rule-based heuristics using this measure on real data. Figure 4.2(b) plots this ratio for both methods. The rule-based heuristic ranges from 50% to 100%, with an average of 80% while Exhaustive had an average ratio of 20% with about 20 strings having ratio 0. In fact, Exhaustive beat rule-based on all the 100 strings except one. From these experiments, we conclude that our edit distance based approach succeeds in reverting the strings towards their original form, far more than the rule-based heuristics.



(a) Tree Edit Distance **(b) String Edit Ratio**

Figure 4.2: Goodness of Exhaustive and Rule-Based Repairs

We also tested our approach using data from *Web Repository*, which contains real malformed XML data. Since no gold standard of what was intended by the creator of any of these XML documents was available to evaluate goodness of repair, our experiments compare the number of operations to obtain a well-formed string using our methods with that obtained using the rule-based heuristics. With number of tags ranging from tens to thousands, Figure 4.3 presents the results on real data. With few errors in the string, our approach was only marginally better than the rule-based. However, there was up to an order of magnitude difference for strings requiring at least 10 edit operations to repair.

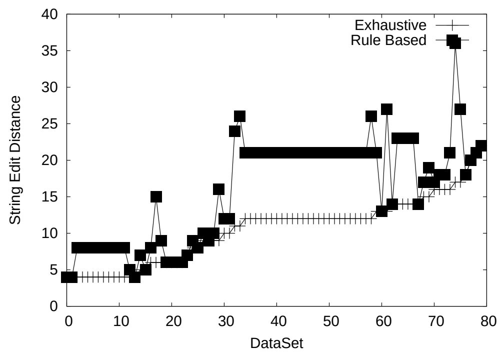
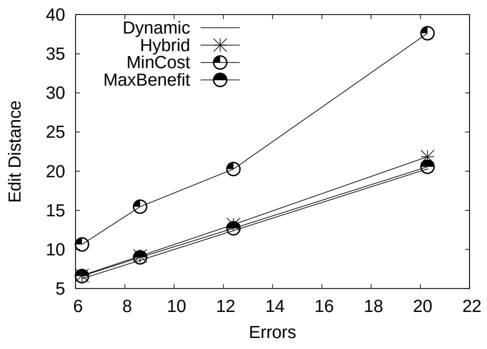


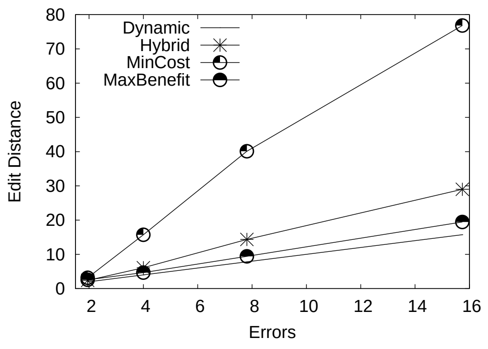
Figure 4.3: String Edit Distance with Real Errors

4.5.3 Single Repair

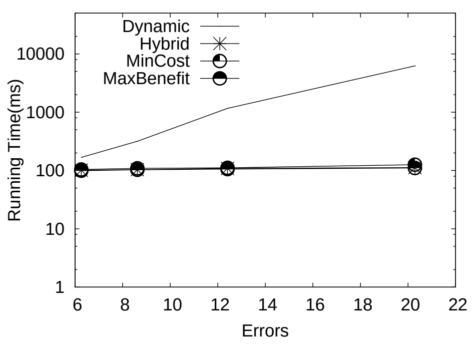
We ran experiments using the dynamic program for repairing documents with text (satisfying $G_{T,W}$ rather than G_T) as well as the following analogues of tags matching(or tags-only) methods: MaxBenefit, MinCost and Hybrid. Figure 4.4 presents performance as a function of number of errors, ranging from 6 to 20 (with fixed length 40,000) on *BGP* and 2 to 16 (with fixed string length 4,000) on *Tree Bank*. DP again is the slowest while MinCost is the least



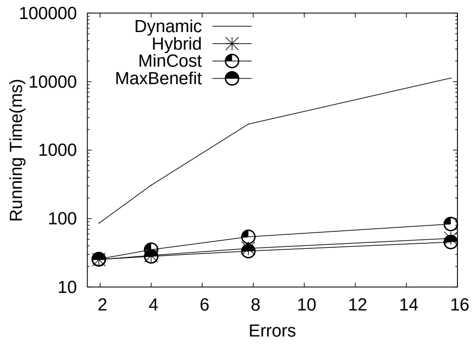
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



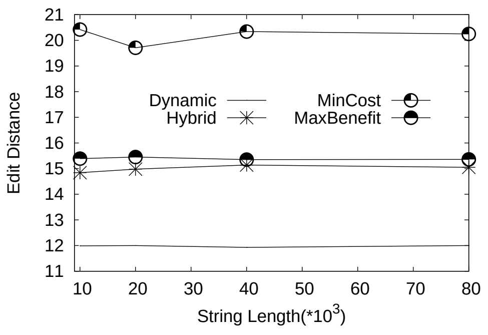
(d) On *Tree Bank* data

Figure 4.4: Single-Repair, *Error Number*

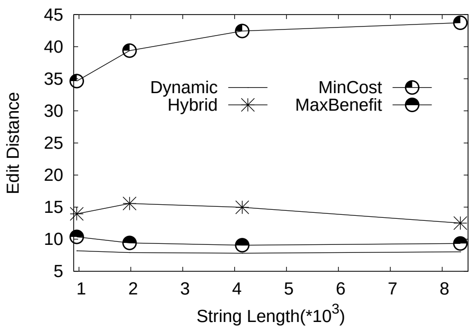
accurate. MaxBenefit is again both more accurate and faster than MinCost and Hybrid is slightly faster but less accurate than MaxBenefit. With the well-formed substring removed, the average string size in Figure 4.4 decreases from 40,000 to $100 \sim 400$ on *BGP*, and from 4,000 to $60 \sim 500$ on *Tree Bank*. The running time for DP grows quickly, due to the increase in string length, while other methods are less affected by errors. MaxBenefit approximates DP well in edit distance and is faster by up to two orders of magnitude.

4.5.3.1 Retaining Well-formed Substrings

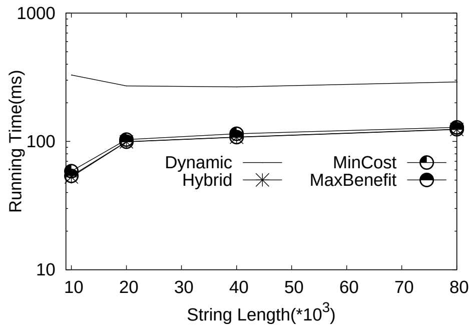
As proved in Sect 4.3.1, the DP algorithm, without well-formed substring elimination, is 2-approximate optimal. To study its effects on accuracy and running



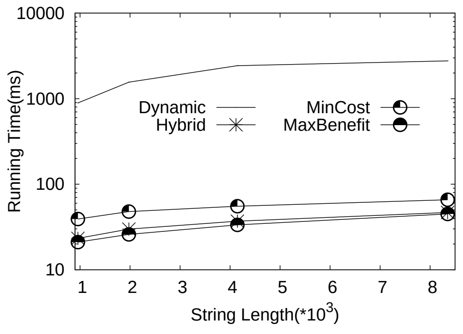
(a) On *BGP* data



(b) On *Tree Bank* data

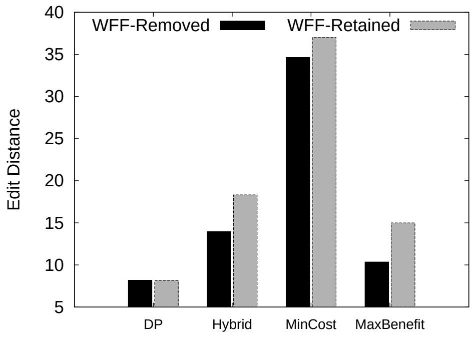


(c) On *BGP* data

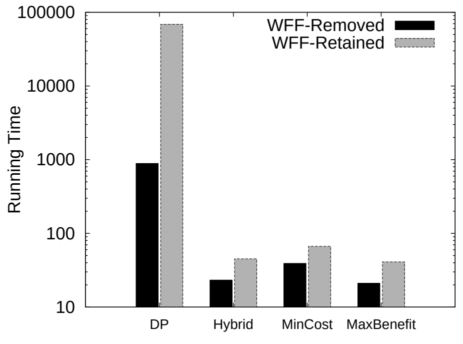


(d) On *Tree Bank* data

Figure 4.5: Single-Repair, *String Length*



(a) Edit Distance



(b) Running Time

Figure 4.6: Well-formed Substring Removal

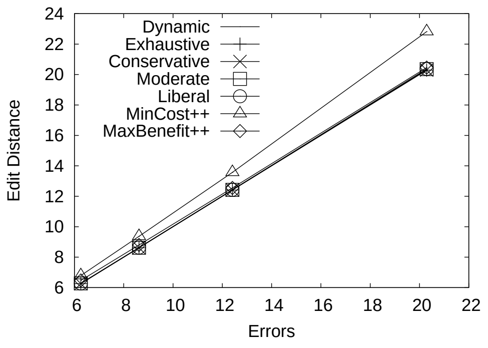
time, we pick one *Tree Bank* dataset with average string length $1,000$, average error number around 8. Figure 4.6 exhibits the relative performance with and without well-formed substring pruning.

There is an insignificant 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. Recalling that these methods are randomized, they may be partly responsible for part of the difference. With a longer string, it takes much more time to build the suffix stacks, leading to the increase in running time. From these evidences, we conclude that well-formed substring is a useful pruning strategy in speeding up the algorithm with little accuracy loss.

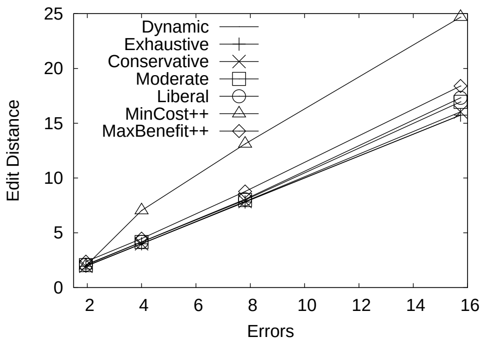
4.5.4 Multiple Repairs

Single-Repair Performance: Figure 4.7 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.

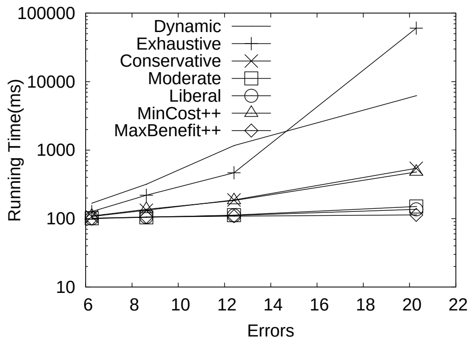
Figure 4.8 illustrates the scalability versus string length with roughly 12 errors for *BGP* and 8 errors for *Tree Bank*. Not surprisingly, Conservative was the second best after Exhaustive. MaxBenefit++ was superior to MinCost++ on *Tree Bank* but not on *BGP*, which shows that the heuristics they're based on are complementary. After pruning, the string length remains around 150 for



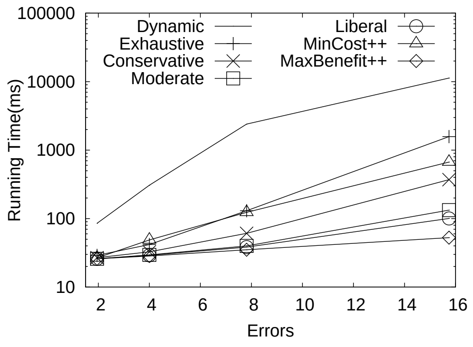
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data

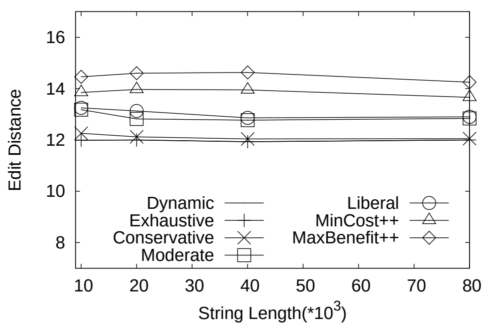


(d) On *Tree Bank* data

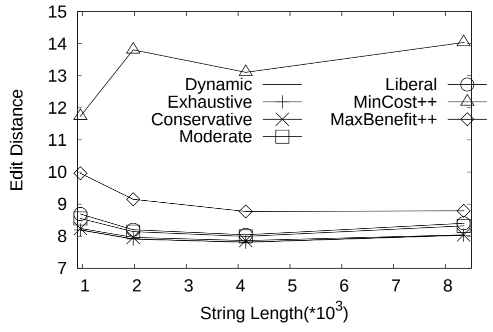
Figure 4.7: Multi-Repair, *Error Number*

BGP 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*.

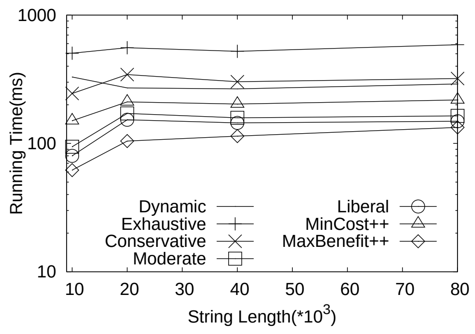
Multi-Repair Performance: Figures 4.9 and 4.10 give the performance and s-calability of branching methods for finding 5 repairs. Again, MinCost++ consistently had the worst accuracy. On *Tree Bank*, Conservative beats MinCost++ on both accuracy and speed, which shows the MinCost heuristic, does not work 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 is fairly constant with increasing string size. The average edit distance



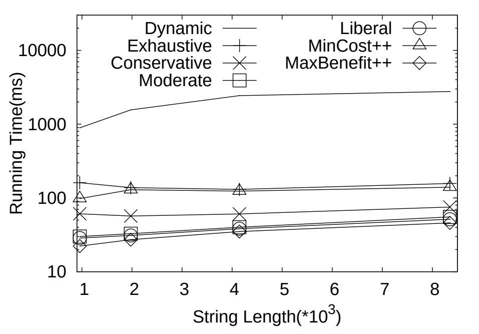
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data

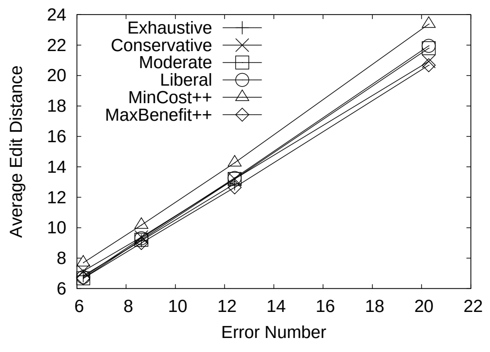


(d) On *Tree Bank* data

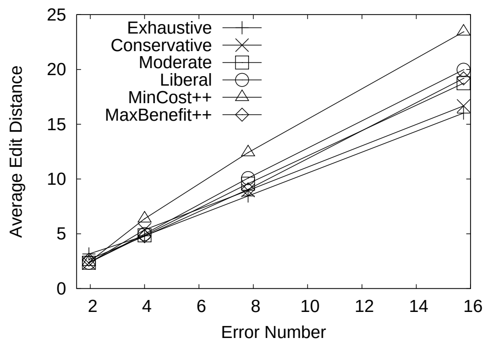
Figure 4.8: Multi-Repair, *String Length*

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.

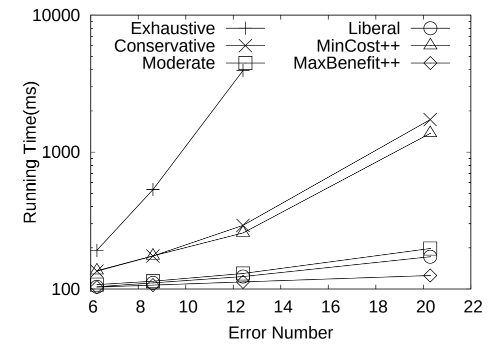
K-**Repair Performance:** We issued queries to find K repairs for K between 1 to 20; the results are shown in Figure 4.11. The methods that prune more aggressively failed to return as many as K repairs in some instances. Only MinCost++, Conservative and Exhaustive were capable of returning K repairs. 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



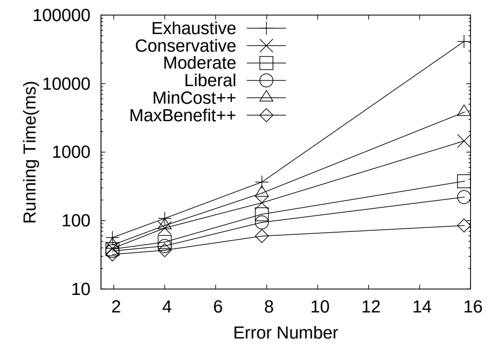
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



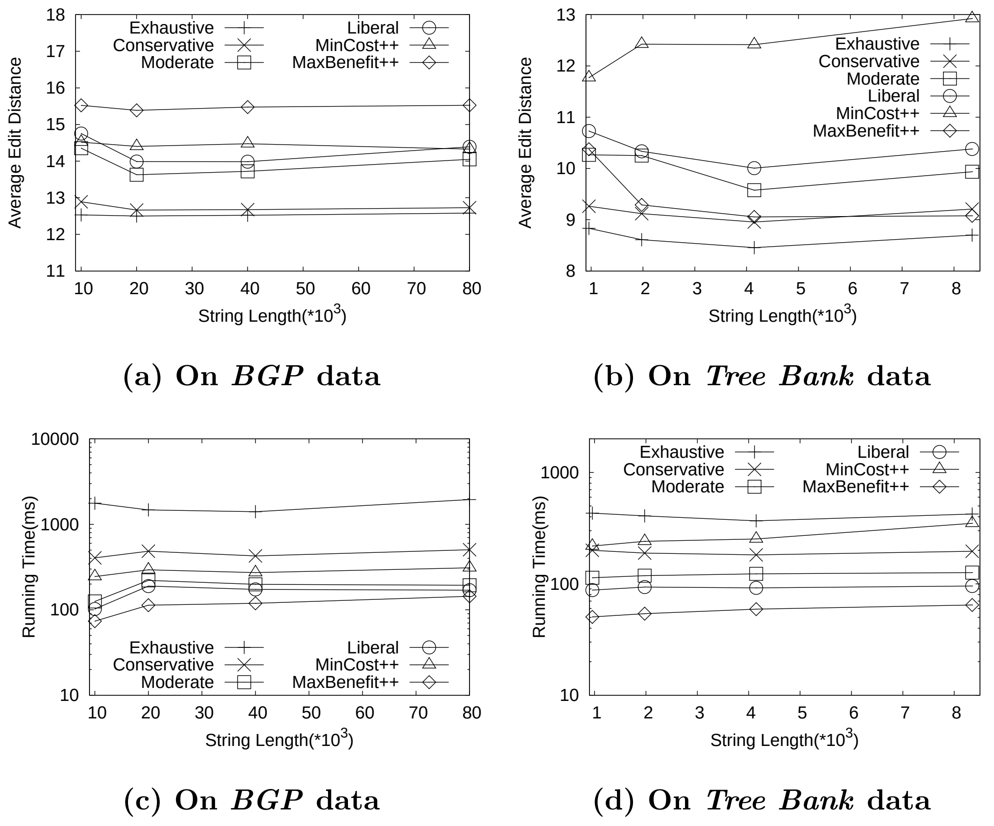
(d) On *Tree Bank* data

Figure 4.9: *top-5* Repairs, *Error Number*

average edit distance grew slowly while the running time grew linearly, which indicates some degree of scalability.

4.6 Conclusions

In this chapter we introduce the algorithms on repairing documents with a more restricted constraint, where text must be embedded within a pair of matching brackets(tags). Such constraint is the most commonly used constraint for semi-structured documents. With minimal edit distance as the optimize goal, most of the strings are repaired towards the right directions, proved by our experimental results on examining the number of errors correctly undone. However, there



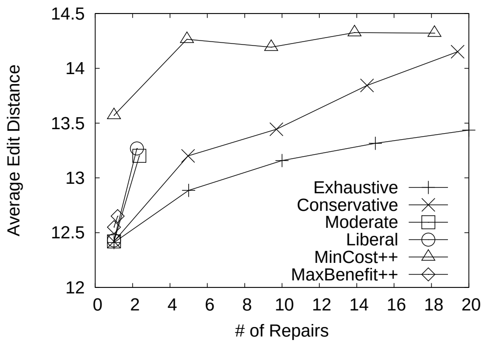
(a) On *BGP* data **(b) On *Tree Bank* data**

(c) On *BGP* data **(d) On *Tree Bank* data**

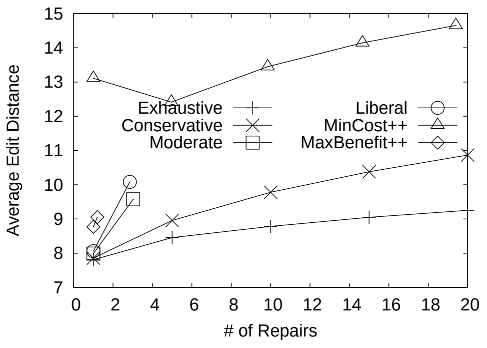
Figure 4.10: *top-5* Repairs, *String Length*

are still cases when the repair may be distant from the original string, as there could be some hidden patterns in the documents.

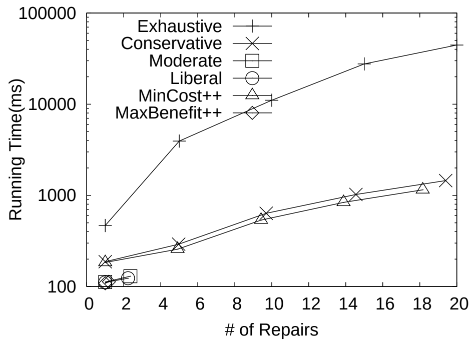
With no companying schema, the two heuristics we use are based on local benefit and reduction in edit cost, which are far from enough. Inferring, from the document at hand, the intention of user or the whole picture of the documents, is impossible. One possible extension to achieve a better result is by mining the hidden patterns, such as frequent q-grams, or frequent sub-trees, which can be extracted from the well-formed sub-strings.



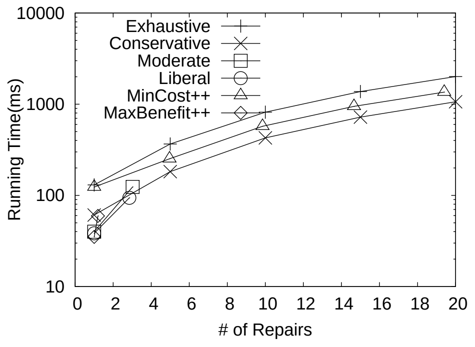
(a) On *BGP* data



(b) On *Tree Bank* data



(c) On *BGP* data



(d) On *Tree Bank* data

Figure 4.11: *top-k*, Scalability

Chapter 5

DETECTING STRUCTURAL ANOMALIES WITH EXPLANATIONS

Even though a document is well-formed in terms of tags, it is not a guarantee of error-free. Many works have been contributed to semantic validation of semi-structured documents, but few focus on the basic syntactic study. In this chapter, we concentrate on detecting a prevalent structural anomaly in semi-structured data, which we refer as unexpected element error*. Unexpected element error occurs whenever there are missing or spurious elements in the data. We propose novel techniques to detect unexpected element errors through a controlled exploration of a lattice structure. Our method also provides plausible reasoning for every reported error and a summarization technique based on variations of set cover for concise reporting. We conduct extensive experiments on several real data sets to verify the usefulness of the proposed techniques. Finally, we present an online visualization tool to assist interactive detection of anomalies.*

5.1 Motivation

The emerging of semi-structured data brings in flexibility and easy customization. A normal user could learn how to construct one with some basic training. Enormous works have been done on detecting data inconsistency in relational database. Functional dependency, inclusion dependency, and conditional functional dependency [39, 38] are some representatives of the kind. But few efforts from the database research community have been devoted to cleaning the structural errors. In general there are two schools of structural errors: mismatched tags(caused by tag missing or tag interleaving), and unexpected elements(such as elements not defined under its parent, or repeated occurrence of elements). Mismatched tags stop the document from being parsed, and unexpected nesting breaks the intention of the document, which is likely to be a sign of incomplete or duplicated data.

Errors from the first school have been thoroughly studied in previous chapters. Errors of the second type are prevalent in real life as well, but not extensively studied. For instance, in the DBLP dataset, we get some `proceedings` with multiple `year`, which turned out to be duplicated records; and some `inproceedings` with multiple `editor`, which are expected to be `author`. Such phenomenon exists in many other widely used datasets, like Mondial, Uniprot,etc [77]. While querying these noisy datasets, such structural errors may give incorrect answers misleading the users and take extra query processing time. Pointing out such un-expectations can help improve data quality and reveal to user the hidden constraints in the document that s/he may not aware of.

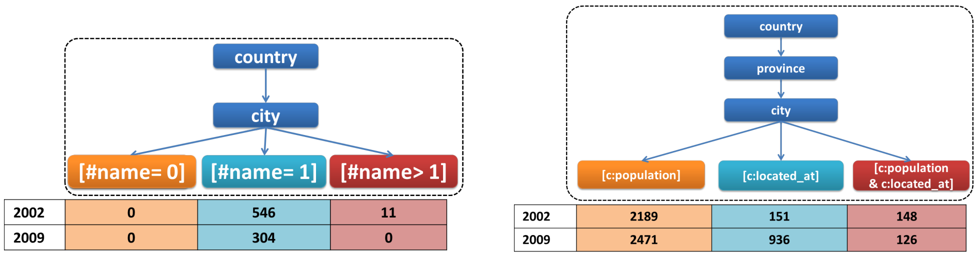
Schema, such as DTD (Document Type Definition) and XSD (XML Schema Definition), designed to define document's expected structured, can be used for document validity verification. To some extent, schema can help alleviate the pain. But schema only is far from satisfactory, for three reasons. First, schema could be oversimplified and cannot express all the structural constraints exactly. While writing the document, the user may have some rules of expertise in mind, but not explicitly expressed in schema. E.g. in the DBLP dataset, most users are aware of the difference between an `author` and an `editor`: an `inproceedings` has multiple `author` and a `proceedings` should have some `editor` , though it is not explicitly restricted by the DTD. But there are few cases, where the `author` sub-element takes the place of an `editor`, and vice versa, due to the inconsistent understanding over the hidden rules. Second, data and schema are seldom updated simultaneously. As data changes quickly, imposing schema on data would be too expensive since schema should be updated accordingly, incurring lots of maintenance efforts. As in DBMS, schema is usually designed ahead, when more data are inserted, there come new requirements and we should avoid shoehorn the new data into obsolete schema. Third, only a small portion 24% [49] of the semi-structured documents online are associated with schema. While dealing with the remaining 76% documents, there is no rule to verify against. Inferring one such schema seems to be an alternative at the first glance. But non existing works on schema inference [19, 19, 18] seriously notice the price of errors. They either assume the data is clean or simply denote those low frequent elements as error and ignore them.

Considering the large scale of data and complexity in schema design, the alternative of consulting a domain expert is far from practical. Hence we need a data-driven approach to be used during the data quality management process. In the following sections, we will show elaborate examples from real world and later present how to capture such errors.

5.1.1 Motivating Examples

"Not expected elements" and "Element content does not follow the DTD" are the main reasons leading to the invalidness of documents, while checked against XSD and DTD respectively [49]. However, we cannot guarantee that those following the schema behave as expected. For this purpose, we conduct a thorough study against one real dataset, *Mondial* . In the following section, we illustrate some interesting findings, serving as our motivation of this work. We take two copies of the dataset published at different time, showing their changes over the time. The two versions for Mondial selected are those published in 2002 and 2009 [1]. Mondial dataset is compiled from several geographical Web data sources, describing geographical data such as countries, mountains and seas.

From the view of structure, we only examine the occurrence of elements, and illustrate the result in Figure 5.1. Each figure consists of two parts: the query tree in dashed rectangle and a table at the bottom. The query tree filters the elements we need and the table shows the difference for the two datasets, with the row headers specifying the corresponding year of data.



2002	0	546	11
2009	0	304	0

2002	2189	151	148
2009	2471	936	126

(a) Intra-Element Inconsistency (b) Inter-Element Inconsistency

Figure 5.1: Suspicious Elements in *Mondial*

The query tree in Figure 5.1(a) is of height three. The leaf nodes are labeled as `[#name op v]`, denoting filtering condition on frequency of `city`, whether

[1]`http://www.dbis.informatik.uni-goettingen.de/Mondial/`

it is equal($=$), or larger($>$) than a value v. The query selects a set **S** of `city` elements under `country`, and further puts each element $s \in S$ into one of the three groups, with frequency of `name` equals to 0, 1 or larger than 1. From the table, we can see that for both datasets, only a small portion of `city` elements have multiple `name` compared with other two groups. We check the value of these elements, and find 9 of them are either abbreviation or duplicated values in different languages. But for the Mondial 2009, all `city` have exactly one `name` as expected.

Figure 5.1(b) gives another example on how potential errors can be detected. The query tree on the top searches `city` with different children: `population`, `located at`, and a mixture of the two. Here the label `[c::`*elem*`]` denotes a predicate or filter condition: having some **child** named *elem*.

Counting the number of elements containing either `population` or `located at` alone dose not reveal anything special, as shown in the first two columns. From the third column, however, an interesting observation arises: in Mondial 2002, the two elements are very likely to present together whenever there is a `located at`. So the 3 `city` contains `located at` but no `population` are suspicious here. But for the Mondial 2009 dataset, the number of city containing `located at` increases significantly while the number of city containing both drops. The reason is two-fold. The first cause is the updates in the Mondial schema from 2002 to 2009. Many cities under `country` have been migrated to a `province` element, which is also reflected in Figure 5.1(a), the number of `city` under `country` drops from 557 to 304. The other season is the population information is not added any more since 2002, while the `located_at` information does.

The above two examples shed lights on how to detect errors: First, skewed distribution on number of occurrence could be a good sign of erroneous elements, where the minority or rare cases could be the errors. Second, the existence of sibling elements could affect the distribution and reveals more potential problems.

5.2 Problem Statement

A well-formed semi-structured document, consisting of nested data, can be naturally abstracted as a single document tree [2]. Multiple document trees can be merged into a single tree by connecting them to a virtual root. For XML documents, each node in the tree corresponds to one element in the document. We use label t_{e_i} for element e_i associated with a node in the tree.

Given a document tree D, our goal is to detect all structural anomalies, and explain these anomalies concisely.

We use standard XPath semantics to denote elements and paths in the document tree. For example, $book[child :: editor]$ selects all elements `book` in the tree that have at least one `editor` as a child, $child :: book[child :: editor]$ selects all elements `book` that are children of the current node in the tree and have at least one `editor` as a child, and $child :: book[child :: *]$ selects `book` which are again children of the current node and have at least one child etc.

A structural anomaly must exhibit certain deviation from expectation, when considering appropriate conditional distribution. In this paper, we capture a large class of conditional distributions (details in Section 5.3) where we observe the frequency distribution of a target (which is a simple element) under a con-

[2]Well-formedness implies data are properly nested with matching beginning and end *brackets* (could be tags in XML, or square bracket in JSON or \begin(...)/end(...) commands in Latex

text (which can be complex). For example, considering Figure 1.3, we would want to compute the frequency distribution of the target `province` under the context of "`country` with at least one child as `state`". If we observe a frequency distribution that is heavily skewed, then that might be an indication of anomalous behavior. For example, if the obtained frequency distribution for element `province` is 200 vs 1 for frequency of 0 and ≥ 1 respectively then this skew strongly suggests that the only element `country` having both `state` and `province` as children is erroneous. The triplet comprising of the context path, target tag and frequency distribution together serves as an explanation for reporting this error: "a country may not have both state and province underneath it". Refer to Figure 5.2 for a pictorial depiction.

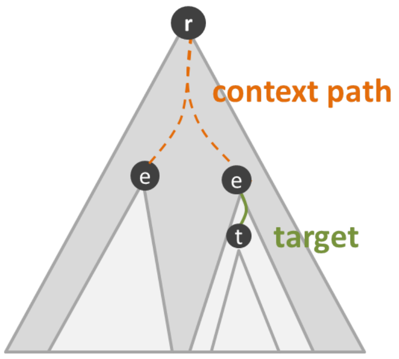


Figure 5.2: Explanation for Structural Anomaly: the context path selects a collection of nodes e and frequency distribution $\mathcal{F} = \{f_{i_1}, f_{i_2}, ...\}$ is computed based on how many times a target element t appears as a child of e, where f_{i_1} represents there are f_{i_1} elements satisfying the context such that each has i_1 children with target tag t.

Definition 5.1 Explanation(ep)

Given a document tree D, an explanation is a triplet

$$\phi = (c, t, \mathcal{F}),$$

where c is the *context* path, t is the *target* tag, and $\mathcal{F}$ represents the distribution of occurrences of elements with tag t as children of nodes satisfying the *context*.

Definition 5.2 Structural Anomaly

Given an explanation $\phi = (c, t, \mathcal{F})$ and a skew threshold $0 < \alpha < 1$, elements that contribute to frequency f_{i_x} where $f_{i_x} < (\sum_x f_{i_x}) * \alpha$ are called structural anomalies due to ϕ given α.

All the structural anomalies identified due to explanation ϕ for a threshold α as $\mathcal{A}_{\phi,\alpha}$. When α is fixed, we will often omit α from the subscript and use $\mathcal{A}_\phi$ to denote the set of anomalies due to ϕ.

Example 5.1 Consider the example in Figure 1.3, and the following *explanation*

$$\phi_1 = \langle country[child :: state], province, \mathcal{F}_1 \rangle$$

Suppose, $f_i < \alpha|\mathcal{F}|$, for $i > 0$, then we expect `country` elements having `state` as children, to not have any child element `province`.

Example 5.2 Consider the following explanations to understand the distribution of `district` under `city`.

$$\phi_2 = \langle country/city, district, \mathcal{F}_2 \rangle$$

$$\phi_3 = \langle country/province/city, district, \mathcal{F}_3 \rangle$$

$$\phi_4 = \langle country/state/city, district, \mathcal{F}_4 \rangle$$

Let $\alpha = 0.01$. From the frequency distribution, we obtain $\mathcal{A}_{\phi_2}$ contains all `city` elements that have 0 occurrence of districts. On the other hand $\mathcal{A}_{\phi_3}$ and $\mathcal{A}_{\phi_4}$ contain all `city` elements that have non-zero occurrence of districts.

5.2.1 Concise Representation for Anomalies

Suppose, we obtain all structural anomalies for a fixed α by exploring each possible explanations. Then returning every such explanation, may be overwhelming for a user to understand. A concise representation of summarization is essential for a better understanding of the anomalies and take appropriate repairing alternatives.

Example 5.3 Reconsider the example in Figure 1.3 and consider the following explanations.

$$\phi_5 = \langle country/province, name, \mathcal{F}_5\rangle$$

$$\phi_6 = \langle country/state, name, \mathcal{F}_6\rangle$$

$$\phi_7 = \langle country/city, name, \mathcal{F}_7\rangle$$

Let $\alpha = 0.01$. Suppose, $\mathcal{F}_5 = \{f_0 = 1, f_1 = 150\}$, $\mathcal{F}_6 = \{f_0 = 2, f_1 = 150\}$, $\mathcal{F}_7 = \{f_1 = 0, f_1 = 100\}$. Clearly $\mathcal{A}_{\phi_5}$, $\mathcal{A}_{\phi_6}$, and $\mathcal{A}_{\phi_7}$ report a province, two states and a city each under country having no name respectively.

Now, consider an alternate explanation.

$$\phi_8 = \langle country/*, name, \mathcal{F}_8\rangle$$

The above explanation has a frequency distribution $\mathcal{F}_8 = \{f_0 = 4, f_1 = 400\}$, and $\mathcal{A}_{\phi_8}$ reports the union of elements reported by $\mathcal{A}_{\phi_5}$, $\mathcal{A}_{\phi_6}$ and $\mathcal{A}_{\phi_7}$. However, ϕ_8 gives a much more concise explanation: "any city, state or province under country must have a name".

Given an α, we refer the elements contained in $\mathcal{A}_{\phi,\alpha}$ as elements **covered** by $\mathcal{A}_{\phi,\alpha}$. Let $\mathcal{A}_{\alpha} = \cup_{\phi}\mathcal{A}_{\phi,\alpha}$. Our goal is to detect a minimum number of explanations to cover all elements in $\mathcal{A}_{\alpha}$. If the number of such returned explanations are still too many, we may want to only return top- k explanations covering the maximum number of anomalous elements. However, this approach needs a close scrutiny.

Suppose, each province and state must have only one `name`, but a city may possibly have two names. For example, Mumbai and Bombay refer to the same city in India. Consider the following example.

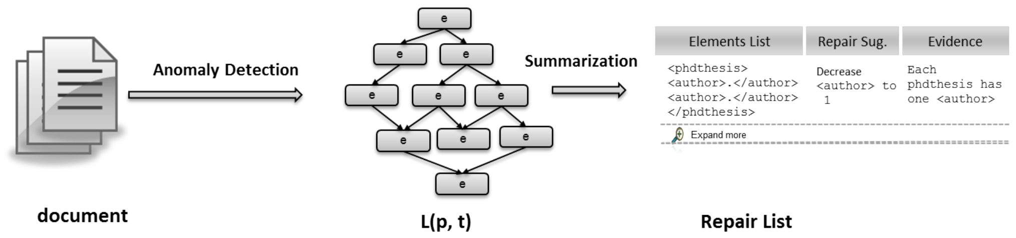
Example 5.4 Set $\alpha = 0.01$ as before and consider explanations ϕ_5, ϕ_6, ϕ_7 and ϕ_8. Suppose $\mathcal{F}_5 = \{f_1 = 500, f_2 = 0\}$, $\mathcal{F}_6 = \{f_1 = 200, f_2 = 1\}$, $\mathcal{F}_7 = \{f_1 = 10, f_2 = 6\}$. Then, $\mathcal{F}_8 = \{f_1 = 710, f_2 = 7\}$. If we report $\mathcal{A}_{\phi_8}$ then it returns 6 false positives. Otherwise, we have to return $\mathcal{A}_{\phi_5}$ and $\mathcal{A}_{\phi_6}$ separately, and that needs one extra explanation to be reported. In the worst case, if we want to avoid all false positives, the number of returned explanations may be large.

We do not want to report too many false positives, on the other hand, avoiding all of them may result in a large number of explanations. To overcome this difficulty, we associate with each explanation a weight, an aggregated score representing the number of false positives that it covers and its description complexity. Note, that if we return an explanation, its weight then indicate how many extra non-erroneous elements need to be examined by a user, and the number of tags used to describe the explanation. So explanations which cover fewer non-erroneous elements and are more concise will be preferred. Given such a weight function and a budget k, we want to return k explanations within total weight W such that they cover a maximum number of truly anomalous elements.

Definition 5.3 Structural Anomaly Summarization

Given a document tree D and a skew threshold $\alpha \in (0, 1)$, suppose all structural anomalies are given by $\mathcal{A}_\alpha$ with associated weights, $w : \mathcal{A}_\alpha \rightarrow \mathbb{I}^+$, and their explanations by $\Sigma = \{\phi_1, \phi_2, ...\}$, then for given parameters k and W, the *structural anomaly summarization* problem finds a set of k explanations such that their total sum of weights is at most W and they together cover the maximum number of elements in $\mathcal{A}_\alpha$.

Figure 5.3 shows the overview of the detection procedure. The first step works on discovering anomalies elements, by designing *Explanation* and a statistical based method to infer the context aware number of occurrence. The second step works on summarizing the explanations, so as to give user a concise but accurate result. Finally, a list of elements is returned to users, with the reason why they are presumed as anomalies.



Elements List	Repair Sug.	Evidence
`<phdthesis>` `<author>.</author>` `<author>.</author>` `</phdthesis>`	Decrease `<author>` to 1	Each phdthesis has one `<author>`

Figure 5.3: Overview

In Sect. 5.3, we discuss how to generate the search space of explanations, and relevant pruning strategies. The *structural anomaly summarization* problem will be visited in Sect. 5.4.

5.3 Structural Anomaly Detection

In this section, we discuss the process of systematically generating a search space to contain all the necessary explanations to detect $\mathcal{A}_\alpha$, and appropriate pruning strategies to keep the computation cost low.

5.3.1 Generating Context Path

The very first step for generating an explanation is to generate all possible context paths. A document tree can be viewed as a rooted labeled tree, where two paths, $p_1 = \langle e_1, e_2, ...e_l \rangle$ and $p_2 = \langle e'_1, e'_2, ..., e'_l \rangle$ are identical iff $t_{e_i} = t_{e'_i}$ for $i = 1, 2, ..., l$ for some positive integer l. Root of the tree has level 0 and level increases by an increment of 1 as we take each step away from the root. It is possible to augment each node in a path with condition c_i where it is additionally required that e_i must have certain siblings (or must not have certain siblings) with labels $t_{x_1}, t_{x_2}, ..$ etc. We use $e_i[c_i]$ to denote such an augmented node, and call such a path, an augmented path. Then two augmented paths $p_1 = \langle e_1[c_1], e_2[c_2], ...e_l[c_l] \rangle$ and $p_2 = \langle e'_1[c'_1], e'_2[c'_2], ..., e'_l[c'_l] \rangle$ are identical iff $t_{e_i} = t_{e'_i}$ and $c_i == c'_i$ for all $i = 1, 2, .., l$. Our contexts consist of distinct augmented paths. Structural anomalies can be captured by looking at the conditional distribution of elements under each such context.

Consider an augmented path $p = \langle e_1[c_1], e_2[c_2], ...e_l[c_l] \rangle$, $level(e_1) > level(e_2) > ... > level(e_l)$. We refer children of e_l as children of path p. Define an indicator random variable X_t taking values $\{0, 1\}$ for each distinct label t and a random variable Y_t again for each distinct label t taking values in $\mathbb{I}^+$ (the set of positive integers). Given p, we let $\mathcal{D}_p(Y_t \mid X_{t_1} = v_1, X_{t_2} = v_2, ..., X_{t_l} = v_l.)$ to denote the conditional distribution of Y_t given an instantiation of $X_{t_1} = v_1, X_{t_2} = v_2, ..., X_{t_l} = v_l$ over the children of paths identical to p. We believe the space of conditional distribution that we consider is (1) sufficient to capture all structural anomalies, and (2) looking at more involved frequency spectrum of several $Y_{t_1}, Y_{t_2}, ...$ conditioned on $X_{t_1}, X_{t_2}, ..., X_{t_l}$ obfuscates the explanation we provide for an anomaly. We allow conditions on each element on an augmented path, except the first one, that is an augmented path must look like, $p = \langle e_1, e_2[c_2], ...e_l[c_l]\rangle$, as most of them starts from the root which has no sibling.

We use a restricted version of XPath semantics to generate the context paths and show that together with target it captures the class of conditional distributions that we consider.

In standard XPath, each path consists of a list of steps $s_1 s_2 \cdots s_m$, where $s_i = axisname :: nodetest[predicate]$.

Example 5.5 Given step $s = country[child :: state]$, it selects all `country` elements that have at least one `state` element as a child. Here $axisname = child$, $nodetest = country$ and $predicate = [child :: state]$.

For generating context path we limit the *axisname* to *child*, and the *predicate* is also limited to *child*. We call this *restricted XPath*.

Lemma 5.4 Restricted XPath along with a target generates all possible conditional distributions of the form $\mathcal{D}_p(Y_t \mid X_{t_1} = v_1, X_{t_2} = v_2, ..., X_{t_l} = v_l)$ where p is an augmented path of the form $\langle e_1, , e_2[c_2], ...e_l[c_l]\rangle$, and $v_i \in \{0, 1\}, i = 1, 2, .., l$.

Proof. Consider any $p = e_1, e_2[c_2]$. To generate $\mathcal{D}_p(Y_t \mid X_{t_1} = v_1, X_{t_2} = v_2, ..., X_{t_l} = v_l)$, consider if $v_i = 1$ then set p_{t_i} as $child :: t_i$ and if $v_i = 0$ then set p_{t_i} as $not(child :: t_i)$. Similarly, if $x \in c_2$ and a sibling with tag x must be present then set p_x as $child :: x$. If a sibling with tag x must not be present then set p_x as $not(child :: x)$. Suppose $x_1, x_2, ..x_s \in c_2$. Use the restricted XPath expression $e_1[p_{x_1}\ and\ p_{x_2}\ and\ ...and\ p_{x_s}]/e_2[p_{t_1}\ and\ ...\ and\ p_{t_z}]$ and use t for the target to obtain the conditional distribution of Y_t given the particular instantiation of $X_{t_1} = v_1, X_{t_2} = v_2, ..., X_{t_l} = v_l$. For any arbitrary length of distinct augmented paths, lemma then follows by induction. □

Example 5.6 The mutual exclusivity of state and a province under country can be represented by considering the conditional distribution $\mathcal{D}_{country}(Y_{\texttt{state}}|X_{\texttt{province}=1})$ and $\mathcal{D}_{country}(Y_{\texttt{province}}|X_{\texttt{state}=1})$. The corresponding restricted XPath expressions are (1) $country[child :: state]$ as context and $province$ as target, and (2) $country[child :: province]$ as context and $state$ as target.

In general, a country may consist of province and/or city. Each province can have districts underneath it. Similarly, each city may also have districts underneath it. However to distinguish between regions marked as city and province, it is unlikely for a city to contain districts when a country has both province and city in it. We can capture the above phenomenon, by the context path $\langle country, city[province]\rangle$ which can be represented as a restricted XPath expression $country[child :: province]/city$, and we are interested in the distribution of $district$.

We let the restricted XPath to contain at most one wild card (*) entry that matches with any label in $nodetest$. Allowing wild card is essential to summarize the anomalies concisely as it allows grouping of explanations, but too many wild card entries can make the explanation vague. We also allow predicates of the form $child :: *$, but there cannot be multiple $nodetest$s such that each of them have predicate of the form $child :: *$. This restriction naturally plays a role how the collection of explanations are generated to form the ultimate search space (Section 5.3.3).

5.3.2 Generating Frequency Distribution

Apart from a context and a target, each explanation contains a frequency distribution $\mathcal{F}$ which is straightforward to compute. Adopting the categorization widely used in DTD cardinality quantifier, we only use $0, 1, \geq 2$ for generating the frequency distribution, that is, it has only three components.

If $\mathcal{F}$ has only one non-zero component, then such a distribution is called *consistent*. On the other hand, if there are multiple non-zero components but $f_i \geq \alpha * |\mathcal{F}|$, $\forall i$, then such an explanation is called α-*non-skewed*. In general, given an α, an explanation is *clean* if it is either consistent or α-non-skewed.

5.3.3 Generating Lattices of Explanations

We now explain the generation of explanations. We generate a single search graph $G = (V, E)$ consisting of possible context paths (restricted XPaths) as vertices. For each target t, we select a subgraph G_t of G to explore, and compute frequency distribution for t given each context $c \in G_t$- this generates the space of explanations for t. We repeat this process for every t.

We consider every distinct node in the document tree as the starting point. Suppose we start from the root of the document tree. We start expanding the context by either exploring vertically in the tree by adding new elements or horizontally by adding new predicates to existing node or by specializing a wild card entry.

We first create a node corresponding to the root of the document tree, and label it by the label of the root. Then we create a directed graph by adding new nodes with the following three types of edges.

- **Horizontal Expansion** Given a node v with label $t_v =$

$e_1, e_2[c_2], e_3[c_3], ..., e_l[c_l]$, we add a directed horizontal edge from v to v' such that v' is obtained from v by either (1) adding (logical and) an additional predicate of the form $child :: elemName$ or $not(child :: elemName)$ with one of the existing predicate, or (2) by adding (logical and) a predicate of the form $child :: *$ with one of e_i that has no predicate associated with it in v only if there exists no other wild card entry in $c_{i'}$, $i' \in \{1, 2, .., l\}/setminus\{i\}$ or (3) by specializing a wild card entry if there already exists one in $e_1, e_2, ..., e_l$.

- **Horizontal Shrinkage** Given a node v with label $t_v = e_1, e_2[c_2], e_3[c_3], ..., e_l[c_l]$, we add a directed horizontal edge from v' to v such that v' is obtained from v by replacing either one of e_i with $*$ if none of $e_1, e_2, ...e_{i-1}, e_{i+1}, .., e_l$ contains a $*$.

- **Vertical Expansion** Given a node v with label $t_v = e_1, e_2[c_2], e_3[c_3], ..., e_l[c_l]$, we add a directed horizontal edge from v to v' such that $v' = e_1, e_2[c_2], e_3[c_3], ..., e_l[c_l], child :: *$ if $e_1, e_2, ..., e_l$ does not already contain a wild card entry.

We claim that the above edge addition process is sufficient to generate all context paths from Section 5.3.1. Further, if we consider each component of the subgraph induced by only the horizontal edges (horizontal expansion and horizontal shrinkage), then each component is a lattice.

Lemma 5.5 The edge addition process (horizontal expansion, horizontal shrinkage, vertical expansion) is sufficient to generate all distinct augmented paths with at most one wild card entry.

Lemma 5.6 Every connected component in the subgraph induced by the horizontal edges form a lattice.

5.3.4 Pruning the Search Space

The above search graph generation process terminates when all distinct augmented paths have been covered. But when exploring this graph for a specific target to generate explanations, often, we do not need to explore this entire search graph and can terminate early. Here we propose two such rules for pruning the search space.

Consistent node in the search graph: Recall that a distribution is said consistent, if it has only one non-zero component. Given an explanation $\phi = \{v, t, \mathcal{F}\}$ if $\mathcal{F}$ is consistent then the node v in the search graph is also consistent. If v is consistent, then we do not explore any horizontal expansion edge from v. The reasoning is clear, since if there is a horizontal expansion edge from v to v' then v' is also consistent. Hence v' will not lead to any explanation that covers an anomaly. Therefore, this pruning strategy does not affect the overall result.

Insufficient support: We expect the error rate to be low–less than α fraction. Hence given a frequency distribution $\mathcal{F}$, the expected number of elements in error is at most $\alpha * |\mathcal{F}|$. If $|\mathcal{F}|$ is sufficiently large, then by the standard large deviation bounds such as the *Chernoff bound*, the probability that the number of errors is much higher than α fraction is vanishingly low. This motivates us to pick all elements that contribute to frequency bucket f_i where $f_i < \alpha * |\mathcal{F}|$ as anomalies.

This reasoning is only valid when $|\mathcal{F}|$ is sufficiently large, otherwise a skew may not necessarily indicate errors. Let us consider a simple example.

Example 5.7 Suppose, $\alpha = 0.01$, $|\mathcal{F}| = 10$ then probability that 1 error occurs is $10 * 0.01 * (0.99)^9 = 0.091$. Hence, even if we see a $1 : 9$ skew, the probability that the element contributing to 1 in the frequency distribution is erroneous is only 0.091. Whereas, if $|\mathcal{F}| = 100$, the same probability becomes $100 * 0.01 * (0.99)^{99} = 0.37$, and hence we have higher confidence of reporting error if we see a $1 : 9$ skew.

Therefore, we use a context-aware counter θ on $|\mathcal{F}|$ such that if $|\mathcal{F}| < \theta$ for an explanation $\phi = (v, t, \mathcal{F})$, then we do not explore the node v using horizontal expansion. If (v, v') is a horizontal expansion edge, and $\phi' = (v', t, \mathcal{F}')$ is the explanation at v' then $|\mathcal{F}'| \leq |\mathcal{F}| < \theta$. Hence the decision to prune the horizontal expansion edges from v is correct.

5.4 Structural Anomaly Summarization

Recall that each explanation $\phi = (c, t, \mathcal{F})$ gives rise to a set of anomalies $\mathcal{A}_{\phi,\alpha}$. The number of explanations covering at least one anomaly may be large. Also, it is possible to cover the same anomalous element by multiple different explanations. For easier verification and repairing anomalies, it is therefore essential to concisely summarize the structural anomalies.

Fix an α, let us denote the search graph by $\mathcal{G} = \{V, E\}$, where V is the set of explanations ϕs and E consists of both horizontal and vertical edges. Any node in this search graph that contains either a wild card entry in $nodetest$ or contains a predicate of the form $child :: *$ helps to group a set of explanations that are connected from it by horizontal edges, but do not add any new true structural anomaly. Hence, in order to generate the universal collection of structural anomalies, we discard any such explanation. Let the remaining of the explanations generate the collection of anomalies $\mathcal{A}_\alpha$ (with abuse of notation).

For each target t, consider G_t, and consider each lattice of explanations in $\mathcal{G}$. They are formed by considering the connected components of G_t induced only by the horizontal edges. The nodes in each of these lattices that have no outgoing edge (horizontal) are called leaves. Note that, they may have a vertical outgoing edge associated with them. First, given the way we have created the search space, if we consider the anomalies covered only by the leaves of the lattices, we cover all of $\mathcal{A}_\alpha$.

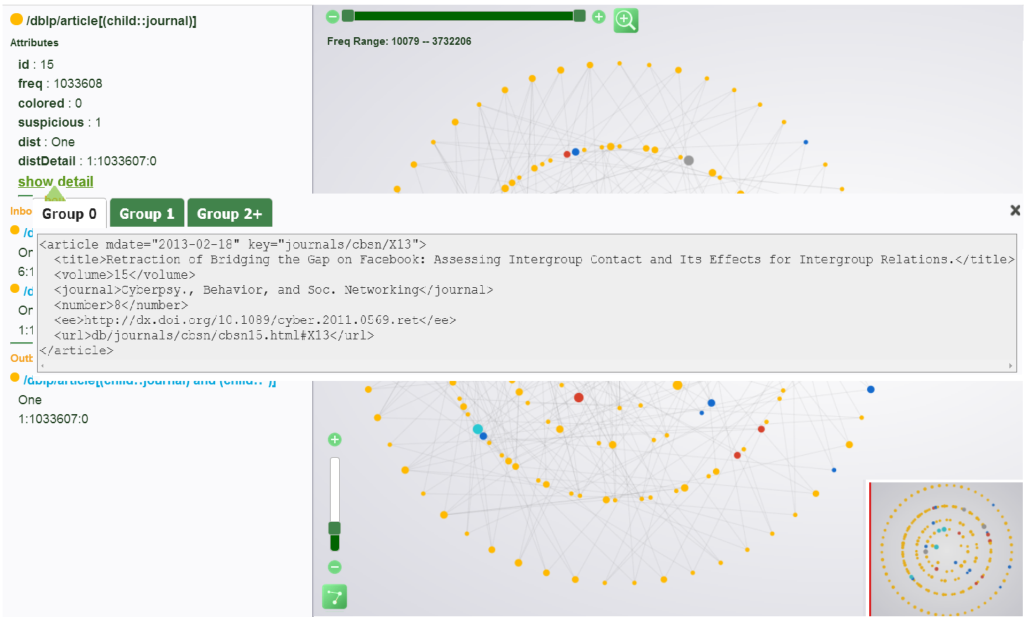
Lemma 5.7 The collection of anomalies covered by the leaf nodes of the lattices induced by the horizontal edges in $\mathcal{G}$ for each target cover $\mathcal{A}_\alpha$.

The above lemma follows from observing that the nodes connected by directed horizontal edges from a node say v also cover all the anomalies covered by v.

As we saw in Section 5.2, while use of explanations with wild card entries help in summarization, they may lead to covering elements that are not structurally anomalous. We therefore want to return a minimum number of explanations that cover as many anomalies as possible, but at the same time do not cover many structurally proper (non-anomalous) elements. Each explanation ϕ is associated with a set of true anomalies that it covers, denoted by $\mathcal{A}_\phi$. The number of non-anomalies that ϕ covers and the description length of ϕ contributes to the weight w of $\mathcal{A}_\phi$. Each non-anomaly associated with an explanation indicates a wasted effort to verify that element, and a longer description length takes extra effort to understand. Since each explanation provides a separate reason for an element to be considered as anomaly, even when an element is covered by more than one explanation, the user will have to judge the applicability of each of the explanations separately. This results in an additive cost function, and we arrive to the problem of *structural anomaly summarization* where given parameters k and W, we want to find k explanations with total weight not exceeding W that cover maximum number of elements in Σ. This is exactly the *size-constrained weighted set cover problem* studied recently in [46].

We first consider a simple setting, where weights are all 0. We show that even when we have a lattice structure, allow at most one predicate, and all $\mathcal{A}_\phi$ have weight 0, still the *structural anomaly summarization* problem is NP-Hard. Not only that, if we want to return a minimum number of rules covering a certain percentage of anomalies, then it is as hard to approximate as the *set cover* problem. Analogously, if we want to maximize the coverage given a fixed number of rules, the problem is as hard as the *maximum-k coverage* problem. Therefore, we can employ the algorithms for the size-constrained weighted set cover problem for general set system over our search graph to summarize the anomalies. We use one such algorithm, namely, *Concise Weighted Set Cover*(CWSC) for our experiments.

The main hardness results stem from the fact that explanations in our search space may generate set of anomalies with arbitrary overlaps. In case, where simplified search space is enough to detect the anomalies, much better results can be guaranteed.



/dblp/article[(child::journal)]

Attributes

id : 15
freq : 1033608
colored : 0
suspicious : 1
dist : One
distDetail : 1:1033607:0

```
<article mdate="2013-02-18" key="journals/cbsn/X13">
  <title>Retraction of Bridging the Gap on Facebook: Assessing Intergroup Contact and Its Effects for Intergroup Relations.</title>
  <volume>15</volume>
  <journal>Cyberpsy., Behavior, and Soc. Networking</journal>
  <number>8</number>
  <ee>http://dx.doi.org/10.1089/cyber.2011.0569.ret</ee>
  <url>db/journals/cbsn/cbsn15.html#X13</url>
</article>
```

One
1:1033607:0

Figure 5.4: An Online Visualization Tool

5.5 A Visualization Tool

In this section, we present an on-line visualization tool to assist anomalies detection. The visualization tool shows a list of explanations after summarization for each dataset, with each row representing an *ep*. To check out the detailed information, user could click on each explanation to see the lattice it involves as shown in Figure 5.4.

The detailed view contains two parts: the lattice in circular layout, and a description of the selected *ep* on the left. In this layout, the inner-most circle is the root and the outer-most circle contains leaf nodes. Users could customize their own layouts such as DAG layout, layered layout for better understanding as they like.

To understand the distribution of expected number of occurrence at the first glance, each node in the lattice is colored based on its expectation: blue (0), yellow (1), red (2^+), etc. The more colorful a lattice is, the more heterogeneous the expected number of occurrence (or the more context-aware the target) is. A more detail description will be given in the left panel by clicking on the lattice node. In the panel, it lists some statistics such as its frequency, observed distribution, inferred expected number of occurrence, and its parent and child nodes.

While those statistics are not enough for users to make a decision, one may prefer to take a look at some samples of the real data before he accepts or decline the anomalies. This function is served by the *Show Detail* button, which queries samples with difference number of occurrence. So users know what is expected and why it is anomalous. In this snapshot, the selected *ep* node discovers 1 `article` with a `journal`, missing `year`. By showing the samples, we find that the article titled "Retraction of Bridging the Gap on Facebook: Assessing Intergroup Contact and Its Effects for Intergroup Relations", is not a journal paper, but a retraction decision on that paper, which should be removed.

With such tool, we believe users could gain a better understanding of the dataset, and learn faster the right way to repair the anomalies. User may refer to the link on our server for trial [3].

5.6 Experimental Study

In this section, we evaluate the proposed techniques with a thorough study on real datasets. We first conduct a case study on the anomalies and later compare our method with the baseline, and then study the robustness of the algorithms. After that we show the efficiency of size-constrained summarization and the pruning strategies. The code is implemented in Java and all the experiments are

[3] `http://db128gb-b.ddns.comp.nus.edu.sg/shanshan/gephi/`

executed on a server with a Quad-Core AMD Opteron(tm) Processor 8356@1 GHz, 128 GB RAM. All datasets are stored in BaseX [1].

5.6.1 Experiment Setup

Dataset : The number of publicly available XML corpora is quite limited. We refer to the XML Repository [4] for datasets published at an earlier age. Four real dataset are used here: two from Mondial (published in 2002 and 2009 resp.), and two from DBLP (published in 2002 and 2013 resp.). The reason we choose Mondial and DBLP is that both collect data from the real world and we could find more facts to verify the correctness of the anomalies reported.

Baseline : To testify the superiority of the way an explanation is defined, in the baseline algorithm, the context is restricted to a single step, and no predicate is allowed. Besides, no summarization algorithm is applied. To distinguish the two algorithms, we call the explanation in the baseline as *b-ep* and the one used in our detection algorithm as *ep*, and our anomaly detection algorithm as *ep-based*.

Metric : To evaluate the quality of anomalies detected, we mainly use two metrics: *precision*, *recall*.

Since no ground truth is given, we conduct a user study on the anomalies to tag them as true error or not. In generally, an anomaly is assumed to be a true error (true positive), if it belongs to one of the cases: 1) Duplicated Elements. Multiple sub-elements are of the same label and same text value, then one of the duplicates should be removed. 2) Missing Element. Certain sub-elements are missing due to incomplete data collected. 3) Invalid Data. The text value of an element violates the intension of that element, e.g., we cannot use the

[4]http://www.cs.washington.edu/research/xmldatasets/

`editor` under `inproceedings` pointing to the authors of a paper.

5.6.2 A Case Study

We perform a case study on the DBLP and Mondial datasets, to show the effectiveness of the techniques. The DBLP dataset records computer science bibliography, including journals, proceedings of conference, books, etc. We refer to the two versions of datasets as DBLP2002 and DBLP2013 resp., with the suffix denoting the year published. We check the correctness of each anomaly manually against other bibliography repositories, such as Google scholar, ACM Digital Library, and SpringerLink.

DBLP Dataset: DBLP2002 has about 300,000 publications, and each publication has 5 to 6 sub-elements on average. We set the context-aware count $\theta = 2000$, and skewed threshold $\alpha = 0.1\%$. We get in total 25 explanations, summarized from over 3500 *ep*s, reporting 350 structural anomalies. Table 5.1 shows 5 explanations of interests. There are six columns in the table: context, target, number of elements hit by the context (#elem), number of anomalies (#ano) and expected number of occurrence (exp).

Table 5.1: Explanations from DBLP 2002

ID	context	target	#elem	#ano	exp
1	/dblp/article	volume	111,609	105	1
2	/dblp/inproceedings (c::pages)	author	208,193	87	1^+
3	/dblp/* (c::booktitle)	number	215,949	54	0
4	/dblp/inproceedings	crossref	328,853	29	0,1
5	/dblp/inproceedings (c::cite)	crossref	6370	7	1

In DBLP, an `article` refers to a journal paper and it should have a `volume` and a `journal` indicting the venue that it appears in. However 105 `article` are detected missing `volume`, according to the first explanation

$$\langle /dblp/article, volume, \mathcal{F}_1 \rangle$$

With the journal name and the article title, we can easily infer the exact `volume`. By grouping these 105 elements by journal name, we get 7 journals and "Electronic Colloquium on Computational Complexity (ECCC)" is the main contributor to 97 out of 105 anomalies. Other six journals contribute the remaining 8, due to a more interesting reason: *no such journal exists*. For instance, DBLP records that Alonso et al. published one paper in "**submitted** *to IEEE Expert*" in 1997, which should be "*'IEEE Expert*" instead. We believe all the 105 elements are true errors to be fixed.

To represent a conference paper, `inproceedings` is used in DBLP. It is a common sense that each paper should have one or more authors. But 87 `inproceedings` with `pages` are found for missing `author`, according to

$$\langle /dblp/inproceedings[child :: pages], article, \mathcal{F}_2 \rangle$$

These 87 anomalies can be categories into following groups: front-matter (35), back-matter (30), conference papers(22). All the publications in front-matter and back-matter should be re-categorized into `incollection`, rather than `inproceedings`. For the 22 `inproceedings` conference papers, the main reasons of `author` missing is authors of these papers are: 1) the paper published on behalf of a group or company, rather than a single person. It could be the design in DBLP that each `author` refers to a person, which does not hold always;2) the authors are not inserted by mistake. So we take these 87 anomalies as true errors here. This example also shows the advantage of using predicates. By removing the predicate, the portion of articles without author is 0.11%, right exceeding the skewness threshold. Therefore none of these anomalies will be reported without using predicates.

The `booktitle` is associated with proceedings, inproceedings, and boo, while `number` is attached to articles. But 54 elements have both `booktitle` and `number`, according to

$$\langle /dblp/ * [child :: booktitle], volume, \mathcal{F}_3 \rangle$$

These 54 elements come from: 1 `proceeding` and 53 `inproceedings`. But none of the 54 `number` have a valid `number` data. In this example, the advantage of using summarization is played. Instead of listing explanations for inproceedings and proceedings separately, a generalized one is favored for conciseness while keeping the correctness.

Another interesting finding is, in DBLP, `crossref` bridges an `inproceedings` to a `proceedings` it appears in.We find 29 `inproceedings` being anomalous for having multiple `crossref` elements, indicated by

$$\langle /dblp/inproceedings, crossref, \mathcal{F}_4 \rangle$$

with expected number of occurrence being 0 or 1. Among these anomalies, 28 of them contain duplicated `crossref` and the other one has inconsistent `crossref` incorrectly referring to different conferences. Such finding is in line with Michael Lay's claim in [59] that "the proceedings records (and the crossref fields) are missing for a lot of legacy inproceedings". Therefore, `inproceedings` in general has zero or one `proceedings`.

Surprisingly by appending one predicate, the expectation differs. According to

$$\langle /dblp/inprocedings[child :: cite], crossref, \mathcal{F}_5 \rangle$$

we know the expected number of `crossref` is only 1, rather than zero or one. 3 anomalies are detected for missing `crossref` and other 4 for duplicated `crossref`. As `cite` is maintained manually, it is possible that the duplication is introduced during manual editing.

Table 5.2: Explanations from DBLP 2013

ID	context	target	#elem	#ano	exp
1	/dblp/article	volume	1033836	1176	1
2	/dblp/inproceedings (not(cite))	crossref	1311151	602	0,1
3	/dblp/article	journal	1033836	228	1
4	/dblp/article	booktitle	1033836	223	0
5	/dblp/* (not(booktitle & publisher)	editor	2357159	11	0

The DBLP2013 dataset is 10 times larger than DBLP 2002. So we set a higher counter threshold $\theta = 10,000$, but with the same skew threshold $\alpha = 0.1\%$. We get 60 *eps* summarized from over 4,000 *eps* covering more than 9,000 anomalies. In Table 5.2, we selectively show 5 explanations.

This table shares some explanations in common with that in Table 5.1. With more publications inserted, the number of anomalies reported also grows. E.g., 1,000 more `articles` are reported for missing or duplicated `volume`, increased from 105 in Table 5.1. Such increase is contributed by 500 more articles with no `volume` from the journal " Electronic Colloquium on Computational Complexity (ECCC)" published between 2001 to 2006, and by 400 more articles with duplicated `volume` sub-elements. However, 1700 more articles published on ECCC before 2001 do have one `volume` each. It is hard to conclude these 400 elements are true errors or not at the moment, but we believe they are still worth reporting for revealing the changes over time.

cite refers to a citation in an inproceedings but is abandoned later for its high maintaining cost [59]. Among the 6,000 more inproceedings with a cite, a large portion 15% of them have 2 crossref each, which pointing to exactly the same proceedings. But among the 1 million more inproceedings *without* a cite, 602 anomalies are detected with duplicated crossref by

$$\langle /dblp/inproceedings(not(cite)), crossref, \mathcal{F}_2\rangle$$

Without the negative predicate $(not(cite)$, the skewness is disturbed by inproceedings with citations and is not significant enough to be reported.

By looking at the two explanations,

$$\langle /dblp/article, journal, \mathcal{F}_3\rangle$$
$$\langle /dblp/article, booktitle, \mathcal{F}_4\rangle$$

We get 228 article missing journal and 223 with unexpected booktitle. Interestingly, these two *ep*s are highly overlapped with each other, which means the error is caused by misuse of elements: all these booktitle should be replaced with journal.

More anomalies are detected by

$$\langle /dblp/*[not(booktitle\&publisher)], editor, \mathcal{F}_5\rangle$$

, i.e., publication without booktitle and publisher should not have any editor. In another word, inproceedings, article and www should not have any editor. Instead of using three eps, the negative predicates give us the chance to express the same meaning in one.

Compared with DBLP2002, more structural anomalies are detected in DBLP2013 as shown in the tables, and a few of them are fixed over the time,

Table 5.3: Explanations from Mondial 2002

ID	context	target	#elem	#ano	exp
1	/mondial/country /province/city	name	2595	75	1
2	/mondial/country	name	231	7	1
3	/mondial/country	encompa-ssed	231	3	1
4	/mondial/country [child::city] /province/city	populat-ion	1404	2	0,1

leading to the disappearance of explanations. E.g., one `proceedings` containing multiple `title`, has been fixed by split into two independent `proceedings` records in DBLP2013; 2 `article` having multiple `pages`, are fixed by retaining the one with a more reasonable value.

Mondial Dataset: The Mondial dataset is compiled from geographical Web data sources, mainly from CIA world factbook [5]. We refer to the two versions of Mondial published in 2002 and 2009 as Mondial2002 and Mondial2009 respectively. To verify the correctness of anomalies, we use the data from GeoHive and CIA world Factbook for comparison.

Both datasets have several thousand elements, so we set the context-aware counter be small $\theta = 150$ and skew threshold as $\alpha = 5\%$. After summarization, we get 8 explanations from Mondial 2002, and some of them are shown in Table 5.3.

In real life, one city or country could be referred using different names, e.g., Ho Chi Minh City was named Saigon; and Myanmar is also named Burma. However, the DTD for Mondial 2002 allows only one `name`. And the error is detected by the first and second *ep*s, detecting in total 82 elements with multiple `name`.

[5]https://www.cia.gov/library/publications

Table 5.4: Explanations from Mondial 2009

ID	context	target	#elem	#ano	exp
1	/mondial/country /province	area	1531	31	1
2	/mondial/country /province	city	1531	18	1^+
3	/mondial/country (c::gdp_agri)	gdp_ind	165	6	1
4	/mondial/country (c::gdp_agri)	gdp_serv	165	6	1

The *ep*s we shown so far verify that lower frequency is a good indicator of errors but, not it is not always the case. 3 out of 231 countries are reported for having multiple `encompassed`. The three countries, Russia, Turkey, Egypt, do span across multiple continents. So the skewness here is caused by rareness, instead of errors.

Allowing predicate in the intermediate nodes of context could help detect more anomalies. E.g. according to the following explanation

$$\langle /mondial/country[child :: city]/province/city, population, \mathcal{F}_4\rangle$$

we detect 2 more anomalies. The *ep* says for countries have both province and city, each `city` under `province` should contain 0 or 1 `population`. Recording population data from different years leads to the multi-`population` error here.

For the Mondial2009 data, after summarization, 33 explanations are returned describing 199 structural anomalies, some of which are shown in Table 5.4.

31 provinces with zero `area` are detected by

$$\langle /mondial/country/province, area, \mathcal{F}_1\rangle$$

These 31 `province` are from Finland(12) and Norway(19), indicating the poor data quality over these two countries and extra attention should be paid.

Other 18 provinces are reported for missing `city`, by

$$\langle /mondial/country/province, city, \mathcal{F}_2\rangle$$

According to *GeoHive*, all the 18 elements have some cities within it, e.g., its capital city. So user should fill up the missing elements there.

Besides geographical data, each `country` may have some GDP related elements, such as `gdp_ind`(GDP Industry), `gdp_agr`(GDP Agriculture) and `gdp_serv`(GDP Service). From the *eps*

$$\langle mondial/country[child :: gdp\_agri], gdp\_ind, \mathcal{F}_3 \rangle$$
$$\langle mondial/country[child :: gdp\_agri], gdp\_serv, \mathcal{F}_4 \rangle$$

we get 6 anomalies as the presence of `gdp_agr` requires the need of other two, as the three are usually available at the same time.

From Table 5.4, we observe that the Mondial data is highly incomplete. Almost all anomalies are detected for the observed number of occurrence is 0 and expected number of occurrence is 1 or 1^+. The incompleteness is also claimed in the Mondial web site that after restructure the "data still highly incomplete".

Comparing the explanations from both datasets, we notice some anomalies are fixed, e.g. each country, province or city has one and only one name, while others still exist, e.g. the number of `country` with multiple `encompassed` grows from 3 to 5, which are caused by rarely occurred facts.

Except Mondial 2002 these interesting anomalies presented in this case study are, by no means [6], detected by schema. Therefore, we believe our context aware explanations, driven by data, are of great help towards automatic data cleaning on semi-structured documents, and could be a good complement of schema.

[6]The DTD for Mondial 2002 is invalid itself, and the dataset violated the DTD in allowing multiple `name` sub-element

5.6.3 Comparison with Baseline

In this section, we compare the results from the baseline algorithm with ours. Two datasets are used, the DBLP dataset published at 2013 and the Mondial dataset published at 2009. We check for each dataset the anomalies detected by both algorithms, and take the union of the true positives from two algorithms as the ground truth. Then we compare the precision, recall of the two algorithms, and show the difference in the explanations they have.

For the DBLP2013 dataset, we set parameters $\theta = 10,000$ and $\alpha = 0.1\%$ for both algorithms. The baseline returns 48 *b-ep* and finds more than 5,000 anomalies, while our algorithm returns 61 *ep* and covers about 9,000 anomalies. Among these 48 *b-ep*s, 24 are in common with our algorithm and other 24 *ep*s are generalized. E.g., five *b-ep*s in the baseline for target `ee` are summarized into a single *ep* in our algorithm. Besides, our algorithm discovers 20 more *ep*s, with some predicates. For instance, 602 `inproceedings` without `cite` but duplicated `crossref` will be missed in the baseline. More interesting *ep*s with predicates have been introduced previously in Table 5.2.

Taking the union of anomalies from both algorithms, we detect in total 7230 anomalies are true errors. The baseline hits 4630 of the true positives and our algorithm covers all. The precision and recall are shown in Figure 5.5. As more anomalies are detected for optional elements with label like `cdrom` and `note`, the precision in our algorithm decreases a little bit compared with baseline.

For the Mondial 2009 dataset, we set $\theta = 150$ and $\alpha = 5\%$. The baseline returns 13 *b-ep*s with 149 anomalies and our algorithm returns 33 *ep*s covering 199 anomalies. 10 of the 11 *b-ep*s are in common with the result of our algorithm, and the last one is much too generalized than ours. It reports all `city` with multiple `located_at` sub-elements to be anomalies, covering 30

Table 5.5: Comparison with Baseline

	DBLP		Mondial	
	Baseline	ep-Based	Baseline	ep-Based
precision	0.88	0.81	0.52	0.76
recall	0.64	1	0.51	1
#anomaly	5,255	9,006	149	199

more anomalies than our algorithm. This is because in the baseline, it does not distinguish `city` of a `country` from that of a `province`. So the skewness caused by diversified distribution in specialized context will not be checked. Besides, we detect 16 more *ep*s with some predicates in the context discovering more anomalies caused by data incompleteness. Among all the anomalies from both algorithms, we get 151 true positives. The precision and recall are shown in Figure 5.5, which proves the superiority of our algorithm.

5.6.4 Sensitivity to Parameters

In this section, we examine the robustness of the detection algorithm against two parameters: the context aware count θ and the skew threshold α.

The counter threshold controls whether we should stop expanding one node in generating the search graph. The larger the counter, the more explanations are pruned. As we take the anomalies in the leaf nodes as the universal set, the change in θ will affect that set, with some new anomalies emerging and some old anomalies removed. When the threshold is too small, the result may over-fitting and many anomalies reported are not interesting enough. But when the threshold is too larger, many interesting nodes will be pruned off.

The skew threshold α plays a different role. The change in α will not affect number of explanations in the graph, but will change the number of anomalies covered by each *ep*. With a larger α, more elements are detected as anomalies and the universal set will increase monotonically. So the returned number of *ep*s could increase (as some non-suspicious *ep*s become suspicious now, or some suspicious *ep* detect more elements as anomalies), or decrease(many *ep*s are summarized into one).

Metric: Besides precision and recall, we also compare the change in explanation lists. In this experiment, we take the union of true errors under all parameter settings as ground truth. When θ increases from v_i to v_j, we compare the explanation lists $\Sigma_{\theta=v_i}$ with $\Sigma_{\theta=v_j}$. To show the difference in summarized *ep*s, we classify each *ep* into some group in following way. An *ep* in $\Sigma_{\theta=v_i}$ could 1) remain in the new list, (*same*), 2) have a generalized counterpart(*gen*), 3) have a specialized counterpart(*spec*), 4) does not exist anymore(*sink*), in $\Sigma'_{\theta=v_j}$. And we call those *ep* $\in \Sigma'_{\theta=v_j} - \Sigma'_{\theta=v_i}$ as *emerge* explanations. When tuning α, we also perform the similar comparisons on adjacent explanation lists. Table 5.6 lists the values we use for experiments, and default values are in bold.

Table 5.6: Parameter Setting

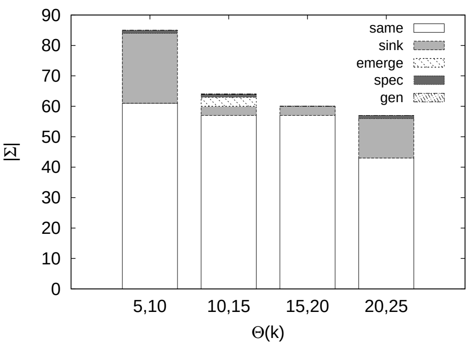
Dataset	θ	$\alpha(\%)$
DBLP	(5, **10**, 15 ,20, 25)*10^3	0.05, **0.1**, 0.15, 0.2,0.25
Mondial	50, 100, 150, 200, 250	1, 3, **5**, 7,9

5.6.4.1 Sensitivity to Context-aware Counter

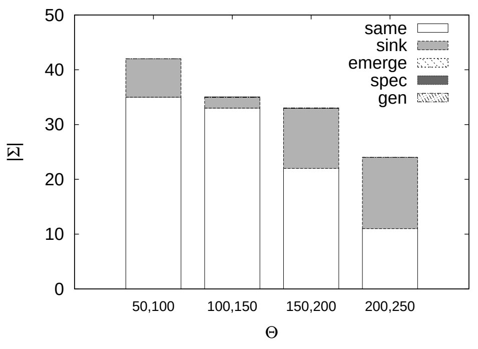
For the DBLP dataset, we tune the counter θ from 5K to 25K, with $\alpha = 0.1\%$. Figure 5.5 shows the difference in explanations. Each bar in Figure 5.5(a) shows the differences between two explanation lists, and the label under each bar indicates two adjacent θ for comparison. From this figure, we can see most *ep*s are shared by two adjacent explanation lists. A couple of *ep*s are sinked due to insufficient cardinality. As expected, there also emerges some new *ep*s.

E.g., in the second bar, we notice 3 new *ep* returned. With a higher counter threshold, some internal nodes in the lattice become leaf nodes, and they form the new universal set.

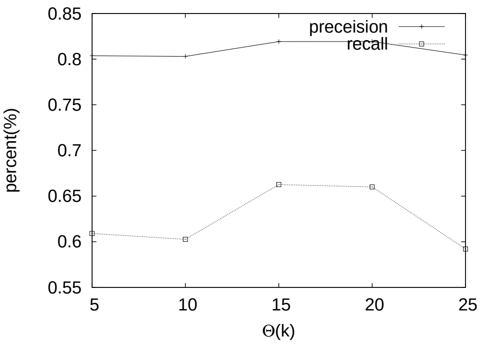
In Figure 5.5(c), we show the precision and recall against $theta$. When θ grows from 5K to 15K, both precision and recall increase. But when θ grows from 15K to 25K, both precision and recall drop, in particular the recall. This is because a smaller threshold may cause over-fitting and detect anomalies with a low confidence, but a larger one prunes too many interesting nodes from the search graph.



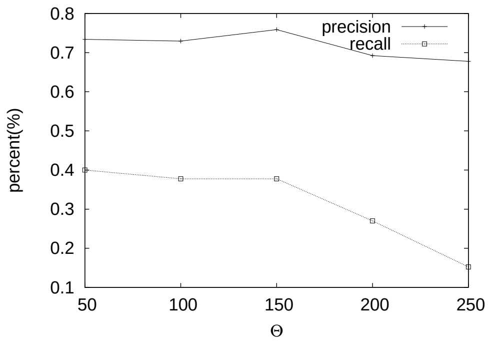
(a) Change in *eps*, DBLP2013



(b) Change in *eps*, Mondial 2009



(c) Quality, DBLP 2013



(d) Quality, Mondial 2009

Figure 5.5: Changes in *ep* and Quality Against θ

For the Mondial dataset, the context aware counter θ is increased from 50 to 250, while the skewed threshold α is fixed at 5%. The comparison of explanations between adjacent parameters is shown in Figure 5.5(b). The figure reveals the same trend as we see in DBLP 2013 dataset: most of the *ep*s are shared by two lists and the sinked eps are pruned due to lower cardinality.

In Figure 5.5(d) we show the precision and recall trends. As Mondial is mainly suffering from "incomplete elements", a smaller θ captures all most all the errors. So the recall drops when θ grows. However the change in precision is a little bit different. This is because when $\theta = 250$, it prunes half of the *ep*s, including both false positives and true positive. So the precision grows when θ grows from 200 to 250, but the recall drops significantly.

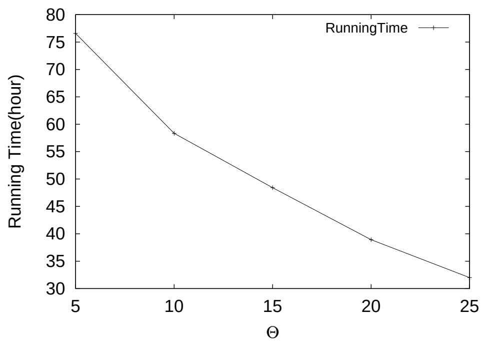
We can see from these results θ should be neither too small nor too large. For the two dataset we used, when θ is about 1% of total elements, it works best.

Figure 5.6 shows the overall running time against θ for both dataset. With the increase of θ, the running time drops quickly, as a larger θ filters out more explanations in the search graph and the query time is notably saved. As the DBLP 2013 dataset is extremely large, it may cost 5 7 second per *ep* to query the occurrence groups. So it may take up to 75 hours to process it. The Mondial 2009 dataset is relatively small but deep in structure, and it costs about up to 5 hours for anomalies detection.

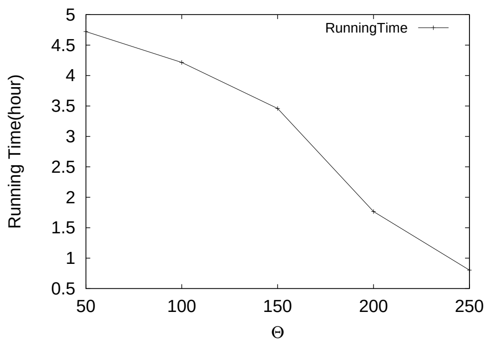
5.6.4.2 Sensitivity to Skew Threshold

The increase in α will not affect the number of nodes in the search graph structure, but will cause the increase in the number of anomalies.

For DBLP 2013 dataset, we fix the counter threshold θ to be 10,000, and tune α from 0.05% to 0.25%. For Mondial 2009 dataset, the counter θ is fixed at 150 and α is tuned from 1% to 9%. The x-axis in Figure 5.7 shows the



(a) DBLP 2013



(b) Mondial 2009

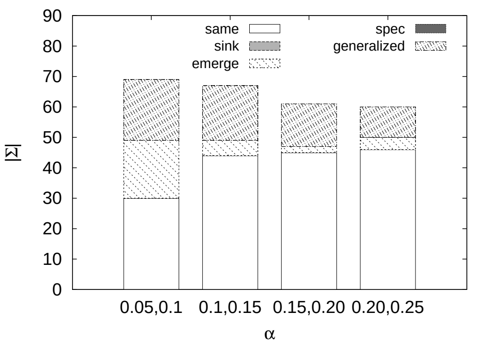
Figure 5.6: Running Time Against θ

two adjacent α values for comparison, tokenized by comma. From these two figures, we can see most of the *ep*s are either the same or generalized, and a small number of new *ep*s emerges.

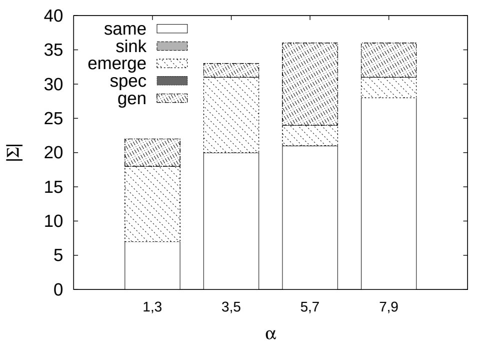
A larger α leads to more anomalies, but not necessary more explanations, as explanations could be replaced by a more general more covering a larger portion of anomalies. So from DBLP2013, when α grows from 0.05% to 0.25%, the number of *ep*s after summarization decreases slightly from 62 to 55.

With α increases, there come more true positives and more false positives as well. So the recall keeps growing while the precision drops significantly after 0.1% in Figure 5.7(c). And the similar trend is observed in Figure 5.7(d). A smaller α gives a high precision but a low recall while a large α gives a low precious and a high recall. To start with, user may use a smaller α to check the result and gradually tune the value to a larger one to see more result. From these figures we can see that the most suitable α values for DBLp2013 and Mondial 2009 are 0.15% and 7% respectively.

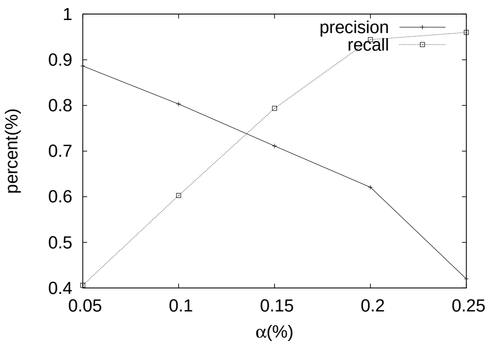
As the change in α does not affect the number of explanations in the lattice, there is almost no difference in running time. So the change in running time against α is not presented.



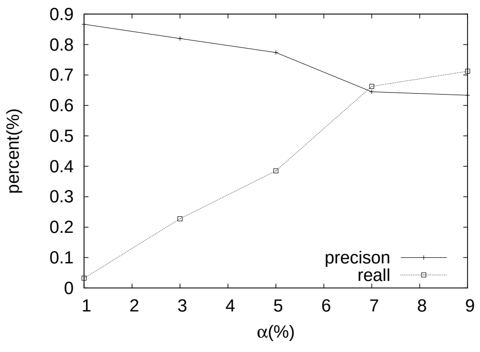
(a) Change in *ep*s, DBLP 2013



(b) Change in *ep*s, Mondial 2009



(c) Quality, DBLP 2013



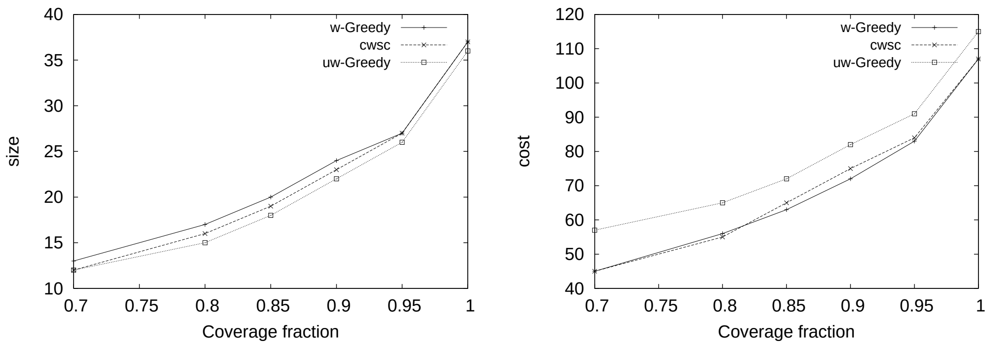
(d) Quality, Mondial 2009

Figure 5.7: Change in *ep* and Quality Against α

5.6.5 Size-Constrained Weighted Summarization

In this section, we show the efficiency of the Concise Weighted Set Cover (CWSC) algorithm, compared with weighted greedy set cover w-Greedy) and un-weighted greedy set cover (uw-Greedy) algorithms. Each *ep* covers some suspicious elements (Benefit), and some non-suspicious elements, compared with the global anomalies. In this experiment, the weight of each *ep* is the **sum** of number of non-suspicious elements and the number of predicates and step in the context path. The w-Greedy and uw-Greedy algorithms use the Marginal Gain

(=Marginal Benefit/Weight) and the Marginal Benefit as heuristics respectively. The summarization is conducted on Mondial2009, as it is more diversified in



(a) Mondial 2009 **(b) DBLP 2013**

Figure 5.8: Size-Constrained Summarization

structure. We first apply w-Greedy and uw-Greedy set cover algorithms to get the number of *ep*s to hit enough suspicious elements, with coverage increased from 0.7 to 1. Then under each coverage value, we turn the constrained size threshold K from 1 to the size needed by w-Greedy algorithm, then reports the K value and the cost when the first solution is found by CWSC. As shown in Fiure 5.8, we find w-Greedy returns a slightly larger size but lower cost, and uw-Greedy covers the set with fewer *ep*s but a higher cost. The CWSC algorithm is just in between these two algorithms, in both size and cost. Given coverage, the size needed by w-Greedy and CWSC differs by one or two. This is because these *ep*s are highly overlapped and the distribution of Benefit is highly skewed. So the advantage of size-constrained weighted set cover is not that obvious. But still, we can see its work on trading off size with cost.

Table 5.7: Pruning Strategies V.S. Search Graph Size

Dataset	Pruning	Consistent	Insufficient
DBLP2013	4363	1263	123
Mondial2009	14914	5479	14271

5.6.6 Pruning Strategies

In this sections, we illustrate the effectiveness of the pruning strategies proposed in Sect. 5.3, performed on Both DBLP2013 and Mondial2009 datasets. In Table 5.7, the first column shows the number of *ep*s needed after pruning, and other two column show the number of *ep*s pruned by different strategies.

For DBLP2013, it prunes off 23% of the *ep*s, mainly by consistency pruning. As the counter threshold $\theta = 10,000$, far smaller than the number of total elements, the insufficiency strategy prunes slightly more than 100 *ep*s. For the Mondail2009 dataset, both consistency and insufficiency contribute a lot and the two prune off over 76% *ep*s. This is because the structure of Mondial is more diversified and many elements have low frequency. From this table, we can see that our pruning strategies do help reduce the search space a lot.

5.7 Conclusions

In this work we focus on detecting one prevalent type of structural errors, the unexpected elements, referred as the structural anomaly. We propose to model the anomalies by an *explanation* and propose novel techniques to detect unexpected element errors through a controlled exploration of a lattice structure. We conduct a thorough case study on real dataset to show the advantage of the detection and perform sensitivity experiments on several real datasets to verify the usefulness of the techniques proposed.

Chapter 6

CONCLUSION

6.1 Summary

In this thesis, we study the poor data quality problem on semi-structured documents from the aspect of structure. We investigate two levels of structural issues: the tag-level problem, when open- and close- tags are mismatched, and the element-level problem, when the occurrence of the elements is unexpected.

Repairing of malformed data at the tag level is based on two variants of well-formedness: tag-only and tag-with-text. 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. Several algorithms are designed targeting at different cases, when the error number is low or high. The efficiency of algorithms on these two variants are verified through experimental study on both real and synthetic dataset.

Detecting the unexpected elements is based on the conditional number of occurrence, inferred from the dataset. Existing work on schema inference works well but ignores the existence of noise. Detecting the unexpected elements is the very first step towards repairing them to improve the quality further. Through extensive experimental study on real datasets, the effectiveness of the strategy is proved, and we believe our method is a complement to schema inference. The latter focuses on schema design, while our constraints are more data-driven. Besides, the online visualization tool we designed would be a great help to anomalies detection.

6.2 Future Work

We are not aware of any extensive study on semi-structured data regarding structural issues, and we believe the repairing and detecting algorithm proposed in this thesis would be of great help to users who deal with semi-structured data. There are still many promising future works to be done.

In Chapter 3 and Chapter 4 we study different algorithms to repair the documents to a well-formed one, targeting at minimum edit distance. The one with minimum edit cost is assumed to be correct and the optimal repair. However, as shown in the experimental study, there are cases, though very few, the data is repaired towards a different direction, widening the distance between the repaired data and the original one. A promising extension would be to study the patterns from the documents. Though failed to be parsed as a document tree, the document still have many well-formed substrings to form many small subtrees. So we can use the set of subtrees along with the frequent q-grams parsed from the document string to guide the repair, which should result in a better repair in tree edit distance.

In Chapter 5 we propose to capture the unexpected elements using the conditional number of occurrence. In that work, we limit our focus on discussing the usefulness of the strategy. More sophisticated inference methods can be integrated to infer the expected number of occurrence accurately, such as histogram based and kernel function based methods. Another interesting extension would be on repair suggestion. In this thesis, we only use a greedy algorithm for summarization, and do not give any valuable suggestions on how to repair. There are many hidden information to be mined. For instance, we notice there are some `inproceedings` with `editor` as children. The two most likely repairs are: substitute `inproceedings` with `proceedings`, or replace `editor` with `author`. If the record has `publisher` and `series`, it is more likely to be a `proceedings`, while the existence of `crossref` makes it more possible to repair the `editor` element. By studying the correlation between elements and the text value, we believe it will produce valuable suggestions on data repairing.

The next extension would be to provide a toolkit, integrating the repair from the well-formedness repair with the unexpected elements detections. So that users in heavy burden of repairing mal-formed semi-structured document will be greatly relieved with few clicks and will be engaged in few places for verification.

BIBLIOGRAPHY

[1] BaseX. http://basex.org. [cited at p. 113]

[2] Beautiful Soup. http://www.crummy.com/software/BeautifulSoup/. [cited at p. 18]

[3] HTML Tidy. http://tidy.sourceforge.net/. [cited at p. 18]

[4] Neko HTML. http://nekohtml.sourceforge.net/. [cited at p. 18]

[5] Trang. http://www.thaiopensource.com/relaxng/trang.html. [cited at p. 24]

[6] XML Beans. http://xmlbeans.apache.org/. [cited at p. 25]

[7] XML Schema Definition Tool. http://msdn.microsoft.com/en-us/library/x6c1kb0s(vs.71).aspx. [cited at p. 25]

[8] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.*, 1(4):305–312, 1972. [cited at p. 18]

[9] Marcelo Arenas, Jonny Daenen, Frank Neven, Jan Van den Bussche, Martin Ugarte, and Stijn Vansummeren. Discovering xsd keys from xml data. In *SIGMOD Conference*, 2013. [cited at p. 2, 25]

[10] Marcelo Arenas, Wenfei Fan, Leonid Libkin, and Leonid Libkin. On verifying consistency of xml specifications. In *PODS*, pages 259–270, 2002. [cited at p. 20]

[11] Marcelo Arenas, Wenfei Fan, Leonid Libkin, and Leonid Libkin. Consistency of xml specifications. In *Inconsistency Tolerance*, pages 15–41, 2005. [cited at p. 20]

[12] Marcelo Arenas and Leonid Libkin. A normal form for xml documents. *ACM Trans. Database Syst.*, 29:195–232, 2004. [cited at p. 22]

[13] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of xml documents. pages 710–751, 2004. [cited at p. 19]

[14] Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, Marcelo Arenas, and Marcelo Arenas. Efficient incremental validation of xml documents. In *ICDE*, pages 671–682, 2004. [cited at p. 19]

[15] Andrew R. Barron, Jorma Rissanen, and Bin Yu. The minimum description length principle in coding and modeling. *IEEE Transactions on Information Theory*, 44(6):2743–2760, 1998. [cited at p. 26]

[16] George Beskales, Ihab F. Ilyas, and Lukasz Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3(1):197–207, 2010. [cited at p. 19]

[17] George Beskales, Ihab F. Ilyas, Lukasz Golab, and Artur Galiullin. On the relative trust between inconsistent data and inaccurate constraints. In *ICDE*, pages 541–552, 2013. [cited at p. 25]

[18] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning deterministic regular expressions for the inference of schemas from xml data. In *WWW*, pages 825–834, 2008. [cited at p. 11, 93]

[19] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise dtds from xml data. In *VLDB*, pages 115–126, 2006. [cited at p. 11, 23, 93]

[20] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of concise regular expressions and dtds. *ACM Trans. Database Syst.*, 35(2), 2010. [cited at p. 24]

[21] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring xml schema definitions from xml data. In *VLDB*, pages 998–1009, 2007. [cited at p. 11]

[22] Geert Jan Bex, Frank Neven, Stijn Vansummeren, and Stijn Vansummeren. Inferring xml schema definitions from xml data. In *VLDB*, pages 998–1009, 2007. [cited at p. 4, 24]

[23] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD Conference*, pages 143–154, 2005. [cited at p. 7, 20]

[24] Utsav Boobna, Michel de Rougemont, and Michel de Rougemont. Correctors for xml data. In *XSym*, pages 97–111, 2004. [cited at p. 4, 18]

[25] Shaofeng Bu, Laks V. S. Lakshmanan, and Raymond T. Ng. Mdl summarization with holes. In *VLDB*, pages 433–444, 2005. [cited at p. 26, 27]

[26] Peter Buneman, Wenfei Fan, Jérôme Siméon, Scott Weinstein, and Scott Weinstein. Constraints for semi-structured data and xml. pages 47–45, 2001. [cited at p. 2, 21]

[27] Fei Chiang and Renée J. Miller. Discovering data quality rules. *PVLDB*, 1(1):1166–1177, 2008. [cited at p. 25]

[28] Fei Chiang and Renée J. Miller. A unified model for data and constraint repair. In *ICDE*, pages 446–457, 2011. [cited at p. 25]

[29] Byron Choi. What are real dtds like? In *WebDB*, pages 43–48, 2002. [cited at p. 64]

[30] Gregory Cobena, Serge Abiteboul, and Amélie Marian. Detecting changes in xml documents. In *ICDE*, pages 41–52, 2002. [cited at p. 9, 64]

[31] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *VLDB*, pages 315–326, 2007. [cited at p. 7]

[32] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang 0001. Nadeef: a commodity data cleaning system. In *SIGMOD Conference*, pages 541–552, 2013. [cited at p. 2]

[33] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. Integrating conflicting data: The role of source dependence. *PVLDB*, 2(1):550–561, 2009. [cited at p. 2]

[34] Xin Luna Dong and Felix Naumann. Data fusion - resolving data conflicts for integration. *PVLDB*, 2(2):1654–1655, 2009. [cited at p. 2]

[35] Xin Luna Dong and Divesh Srivastava. Detecting clones, copying and reuse on the web. In *ICDE*, pages 1211–1213, 2012. [cited at p. 2]

[36] W Eckerson. Data warehousing special report: Data quality and the bottom line. *Applications Development Trends*, 2002. [cited at p. 1]

[37] Wenfei Fan. Xml constraints, 2007. [cited at p. 21]

[38] Wenfei Fan, Floris Geerts, and Xibei Jia. Conditional dependencies: A principled approach to improving data quality. In *BNCOD*, pages 8–20, 2009. [cited at p. 92]

[39] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5):683–698, 2011. [cited at p. 92]

[40] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Wenyuan Yu, and Wenyuan Yu. Interaction between record matching and data repairing. In *SIGMOD Conference*, pages 469–480, 2011. [cited at p. 20]

[41] Flavio Ferrarotti, Sven Hartmann, and Sebastian Link. Efficiency frontiers of xml cardinality constraints. *Data Knowl. Eng.*, 87:297–319, 2013. [cited at p. 21]

[42] Flavio Ferrarotti, Sven Hartmann, Sebastian Link, Mauricio Marín, and Emir Muñoz. Soft cardinality constraints on xml data - how exceptions prove the business rule. In *WISE (1)*, pages 382–395, 2013. [cited at p. 21]

[43] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. Xtract: A system for extracting document type descriptors from xml documents. In *SIGMOD Conference*, pages 165–176, 2000. [cited at p. 23]

[44] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. Dtd inference from xml documents: The xtract approach. *IEEE Data Eng. Bull.*, 26(3):19–25, 2003. [cited at p. 23]

[45] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. The llunatic data-cleaning framework. pages 625–636, 2013. [cited at p. 2]

[46] Lukasz Golab, Flip Korn, Feng Li, Barna Saha, and Divesh Srivastava. Size-constrained weighted set cover with applications. [cited at p. 110]

[47] E. Mark Gold. Language identification in the limit. In *Information and Control*, pages 447–474, 1967. [cited at p. 4]

[48] Sheila A. Greibach. The hardest context-free language. pages 304–310, 1973. [cited at p. 19]

[49] Steven Grijzenhout, Maarten Marx, and Maarten Marx. The quality of the xml web. In *CIKM*, pages 1719–1724, 2011. [cited at p. 3, 4, 9, 12, 30, 93, 94]

[50] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate xml joins. In *SIGMOD Conference*, pages 287–298, 2002. [cited at p. 9, 64]

[51] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *ICDE*, pages 392–401, 1998. [cited at p. 25]

[52] Solmaz Kolahi and Laks V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, pages 53–62, 2009. [cited at p. 7]

[53] Flip Korn, Barna Saha, Divesh Srivastava, and Shanshan Ying. On repairing structural problems in semi-structured data. *PVLDB*, 6(9):601–612, 2013. [cited at p. 10]

[54] Laks V. S. Lakshmanan, Raymond T. Ng, Christine Xing Wang, Xiaodong Zhou, and Theodore Johnson. The generalized mdl approach for summarization. In *VLDB*, pages 766–777, 2002. [cited at p. 26]

[55] Laks V. S. Lakshmanan, Jian Pei, and Jiawei Han. Quotient cube: How to summarize the semantics of a data cube. In *VLDB*, pages 778–789, 2002. [cited at p. 26]

[56] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. pages 1–15, 2002. [cited at p. 19]

[57] Mong-Li Lee, Tok Wang Ling, and Wai Lup Low. Designing functional dependencies for xml. In *EDBT*, pages 124–141, 2002. [cited at p. 22]

[58] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. volume 10, pages 707–710, 1966. [cited at p. 6]

[59] Michael Ley. Dblp - some lessons learned. *PVLDB*, 2(2):1493–1500, 2009. [cited at p. 116, 118]

[60] Leonid Libkin. Normalization theory for xml. In *XSym*, pages 1–13, 2007. [cited at p. 22]

[61] Sebastian Link and Thu Trinh. Know your limits: Enhanced xml modeling with cardinality constraints. In *ER (Tutorials, Posters, Panels & Industrial Contributions)*, pages 19–30, 2007. [cited at p. 21]

[62] F. Magniez, C. Mathieu, A. Nayak, and A. Nayak. Recognizing well-parenthesized expressions in the streaming model. In *STOC*, pages 261–270, 2010. [cited at p. 19]

[63] Yasuhiko Minamide, Akihiko Tozawa, and Akihiko Tozawa. Xml validation for context-free grammars. In *APLAS*, pages 357–373, 2006. [cited at p. 19]

[64] Gene Myers. Approximately matching context-free languages. pages 85–92, 1995. [cited at p. 18, 19]

[65] Gonzalo Navarro. A guided tour to approximate string matching. pages 31–88, 2001. [cited at p. 6]

[66] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *SIGMOD Conference*, pages 419–432, 2008. [cited at p. 26]

[67] Yannis Papakonstantinou, Victor Vianu, and Victor Vianu. Incremental validation of xml documents. In *ICDT*, pages 47–63, 2003. [cited at p. 19]

[68] Mateusz Pawlik and Nikolaus Augsten. Rted: A robust algorithm for the tree edit distance. pages 334–345, 2011. [cited at p. 9, 80]

[69] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, Laurie J. Hendren, and Laurie J. Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *ICSE*, pages 277–287, 2012. [cited at p. 3]

[70] Jason Sankey and Raymond K. Wong. Structural inference for semistructured data. In *CIKM*, pages 159–166, 2001. [cited at p. 23]

[71] Luc Segoufin, Cristina Sirangelo, and Cristina Sirangelo. Constant-memory validation of streaming xml documents against dtds. In *ICDT*, pages 299–313, 2007. [cited at p. 4, 19]

[72] Luc Segoufin, Victor Vianu, and Victor Vianu. Validating streaming xml documents. In *PODS*, pages 53–64, 2002. [cited at p. 4]

[73] Slawomir Staworko, Jan Chomicki, and Jan Chomicki. Validity-sensitive querying of xml databases. In *EDBT Workshops*, pages 164–177, 2006. [cited at p. 18]

[74] Nobutaka Suzuki. Finding an optimum edit script between an xml document and a dtd. In *SAC*, pages 647–653, 2005. [cited at p. 18]

[75] Emily S. Tabanao, Ma. Mercedes T. Rodrigo, Matthew C. Jadud, and Matthew C. Jadud. Predicting at-risk novice java programmers through the analysis of online protocols. In *ICER*, pages 85–92, 2011. [cited at p. 3]

[76] Alex Thomo, Srinivasan Venkatesh, Ying Ying Ye, and Ying Ying Ye. Visibly pushdown transducers for approximate validation of streaming xml. In *FoIKS*, pages 219–238, 2008. [cited at p. 18]

[77] Ba Quan Truong, Sourav S. Bhowmick, Curtis E. Dyreson, and Aixin Sun. Messiah: missing element-conscious slca nodes search in xml data. In *SIGMOD Conference*, pages 37–48, 2013. [cited at p. 10, 92]

[78] Millist W. Vincent, Jixue Liu, and Chengfei Liu. Strong functional dependencies and their application to normal forms in xml. *ACM Trans. Database Syst.*, 29(3):445–462, 2004. [cited at p. 22]

[79] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. pages 168–173, 1974. [cited at p. 6]

[80] Yuan Wang, David J. DeWitt, and Jin yi Cai. X-diff: An effective change detection algorithm for xml documents. In *ICDE*, pages 519–530, 2003. [cited at p. 9, 64]

[81] Xiaoyan Yang, Cecilia M. Procopiuc, and Divesh Srivastava. Summarizing relational databases. pages 634–645, 2009. [cited at p. 26]

[82] Xiaoyan Yang, Cecilia M. Procopiuc, and Divesh Srivastava. Summary graphs for relational database schemas. pages 899–910, 2011. [cited at p. 26]

[83] Cong Yu and H. V. Jagadish. Efficient discovery of xml data redundancies. In *VLDB*, pages 103–114, 2006. [cited at p. 2, 22]

[84] Cong Yu and H. V. Jagadish. Schema summarization. In *VLDB*, pages 319–330, 2006. [cited at p. 26]

[85] Cong Yu and H. V. Jagadish. Xml schema refinement through redundancy detection and normalization. *VLDB J.*, 17(2):203–223, 2008. [cited at p. 2, 22]

[86] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. pages 1245–1262, 1989. [cited at p. 9, 80]

[87] Ning Zhang, Yuanyuan Tian, and Jignesh M. Patel. Discovery-driven graph summarization. In *ICDE*, pages 880–891, 2010. [cited at p. 26]