Repairing Data through Regular Expressions

Zeyu Li Hongzhi Wang Wei Shao Jianzhong Li Hong Gao
Harbin Institute of Technology
lizeyu_cs@foxmail.com wangzh@hit.edu.cn shaowei_acm@163.com
{lijzh, honggao}@hit.edu.cn

ABSTRACT

Since regular expressions are often used to detect errors in sequences such as strings or date, it is natural to use them for data repair. Motivated by this, we propose a data repair method based on regular expression to make the input sequence data obey the given regular expression with minimal revision cost. The proposed method contains two steps, sequence repair and token value repair.

For sequence repair, we propose the Regular-expression-based Structural Repair (RSR in short) algorithm. RSR algorithm is a dynamic programming algorithm that utilizes Nondeterministic Finite Automata (NFA) to calculate the edit distance between a prefix of the input string and a partial pattern regular expression with time complexity of $O(nm^2)$ and space complexity of $O(mn)$ where m is the edge number of NFA and n is the input string length. We also develop an optimization strategy to achieve higher performance for long strings. For token value repair, we combine the edit-distance-based method and associate rules by a unified argument for the selection of the proper method. Experimental results on both real and synthetic data show that the proposed method could repair the data effectively and efficiently.

1. INTRODUCTION

Due to its importance, data quality draws great attention in both industry and academia. Data quality is the measurement of validity and correctness of the data. Low-quality data may cause disasters. It is reported that data errors account for 6% of annual industrial economic loss in America [11]. According to the statistic data of Institute of Medicine, data errors are responsible for 98,000 people's death [17] per year. To improve data quality, data cleaning is a natural approach, which is to repair the data with errors.

When data size is small, it is feasible to repair the errors manually. However, for big data, automatic data repair approaches are in demand. With such motivation, many automatic data repairing methods have been proposed. Three categories constitute such methods. (1) Rule-based repairing methods amend data to make them satisfy a given set of rules [4, 14, 12]. (2) Repair methods based on truth discovery attempt to discover truth from conflicting values [13, 9, 10]. (3) Learning-based repair employs machine learning models such as decision tree, Bayes network or neural network to predict values for imputation or value revision [15, 16, 19].

Constraints or semantics of data are usually represented as rules. Rules for data repairing could be classified into two kinds, semantic rules and syntactic rules. The former one is often described by dependencies among data such as CFD and CIND [6, 7]. Many researches have been conducted on this topic. Syntactic rules are often represented by grammars and used to describe the structure of data. A popular type of grammars is regular expression ("*regex*").

Regular expressions are more suitable for syntactic amendments than dependencies and other semantic rules. Dependencies are defined in semantic domains. It is incapable of solving structure problems such as erroneous insertions and deletions. For example, given $A \rightarrow B$ as a dependency rule, a miswriting of attribute B could be repaired according to attribute A. However, if B is lost, the rule cannot offer instructions on positions to insert B back.

On the contrary, regular expressions can remember essential and optional tokens and their absolute or relative positions. For above example, regex $r = A(xy)^*Bz$ can suggest that the lost B should be added between the repetition of xy and the single z. If A is lost, r can instruct that A's absolute position is at the beginning. Therefore, regex expressions are appropriate choices for regular-grammar-structured data.

Error detection based on regexes has been well studied. Such techniques could tell errors in data by checking whether some components violate the regex. [18] illustrates a spreadsheet like program called "Potter's Wheel" which can detect errors and infer structures by user specified regexes as well as data domains. [21] explains an error detection by regular expressions via an example of 'Company_Name' column from registration entries.

Regex could be used not only to detect errors but also to repair the data automatically. Consider the scenario of a Intrusion Detection System (IDS). IDS generally scans input data streams and matches the malicious code blocks that are harmful to user devices. The properties of such attack are described in the following structure: `<Action><Protocol> <Source IP><Source Port> <Direction Operator>`

Proceedings of the VLDB Endowment, Vol. 9, No. 5

<Destination IP><Destination Port> (<Option$_1$:Value$_1$>;...;<Option$_n$:Value$_n$>) which matches the regex $Act.P.S_{ip}.S_{port} \cdot Op_{dir} \cdot D_{ip} \cdot D_{port} \cdot ([Opt_i : v_i]^*)$. In this regex, each label represents a token with possible values. For instance, "P" can be one of {tcp,udp,ip,icmp}.

If a stream with token sequence <Act><SIP><SP><DO><DIP><DP>(<Opt1:v1>;<Opt2:v2>;) is received, it is known that <P> is lost by matching with the regex. We can repair it by inserting <P> to the token sequence behind <Act> to make it match the regex. Such insertion is a repair.

In addition to the Snort Rule, there exist numerous other instances that are constrained by regular expressions. For example, the MovieLens rating data set contains ratings of approximately 3,900 movies made by MovieLens users [1] and obey the following format:<Sequence Number>::<Movie Name><Year>::<Type$_1$>/<Type$_2$>/...<Type$_n$>/. That is, each movie information is recorded under the constraint of the regular expression $Seq\# :: Name.Year :: (Type/)^*$.

Apart from those two examples, regular expressions have a large number practical applications on constraining data structures from smaller structures like postcode, email address to larger structures like file name matching patterns in Windows System. These examples demonstrate the regex has been widely used. Thus, a regex-based data repairing method is significant and in demand.

Data repairing approaches [12, 6] often make data obey the given rules and minimize the differences between the revised and the original data. Even though we could still follow such idea for regex-based repairing, it is not straightforward to find a proper revision, since the regex-based error detection approaches fail to tell how to make the strings obey the rule.

Edit distance is defined as the minimal number of steps required to convert one string s_1 to another one s_2, denoted by $ed(s_1, s_2)$. It is often used to measure the differences between strings. Since regex-based data repair is to convert a string to the one satisfying some regex, it is natural to adopt edit distance to measure the differences between the repaired data and the original one. Thus, we model the problem of regex-based data repairing via edit distance as follows.

Given a string s and a regex r, find a string $s' \in L(r)$ so that for any $s'' \in L(r)$, $ed(s, s') \leq ed(s, s'')$, where $L(r)$ is the language described by r, i.e. the set of all strings matching r.

To solve this problem, we develop a dynamic programming algorithm. Such algorithm is inspired by Levenshtein distance algorithm but makes improvements for better adaption to the problem of distance concerning a regex and a string rather than two strings. To describe the subproblems for dynamic programming, we use NFA to represent the regex. In the core recursion equation of dynamic programming, previous and next relations of edges in NFA are used as the counterparts of prefixes, which can contribute to the incremental computation of distances between prefixes of s and r. To accelerate the algorithm, we propose an optimization strategy that prunes some redundant computational steps.

Another significant problem in regex-based repair is to select the most suitable candidate of a token from multiple legal choices for insertion or substitution. To address this problem, we propose a decision method so that repaired results can be closer to real values. This method is based on edit distance and association rules which are widely used in data mining field. The final repairing solution is determined by a comparison between the estimated confidence of the distance-based strategy and that of the association-rule-based one.

The contributions of this paper are summarized as follows.

- We use regular expressions for not only error detection but also data repair. We model the problem of data repair as an optimization problem. As we know, this is the first work studying regular-expression-based data repair.
- We develop a whole solution for regex-based data repair with a dynamic programming algorithm as the basic algorithm, a pruning strategy for acceleration, and a proper value selection method to accomplish the repair.
- We conduct extensive experiments to verify the efficiency and effectiveness of the proposed algorithms. Experimental results demonstrate that the proposed algorithms could efficiently repair sequences according to a regex and the quality of the repaired data is high.

This paper is organized as follows. We give an overview of this problem in Section 2 and separate this problem into two subproblems, structural repair and value repair, whose solutions are discussed in Section 3 and Section 4 respectively. We experimentally verified the effectiveness and efficiency of the proposed algorithms in Section 5. At last, we draw the conclusions in Section 6.

2. OVERVIEW

Structural constraints are used to describe the structure that a string should obey. Each structural constraint has two components, token sequence and token value options. The sequence describes the order of tokens in a string and the value options describe the available options of each token. A structural constraint is expressed by a grammar with variables. The grammar describes the token sequence and variables describe token values. We focus on the structural constraint with its grammar described by regexes.

Structural constraints could be used to detect and repair errors in strings. A string s may violate a structural constraint in structural level or token value level. The repair for structural level violation (*structural repair* in brief) is the holistic repair, while that for token value level violation (*value repair* in brief) is the local repair. If value repair is conducted before structural repair, a value revised in value repair may possibly be revised again or even deleted during structural repair. Therefore, structural repair is superior to value repair as the following framework.

1. Repair structural errors by regular expression pattern and construct the repair.
2. Repair value errors inside individual tokens.

As for step 1, the algorithm for structural repair will be discussed in Section 3.

Step 2 aims to choose a proper value for a token from available options. We perform the selection with two approaches based on edit distance and associate rules respectively. The former one finds the value from available options with the minimal edit distance to current value. Such option may be legal since it is the most similar to original value. However, it may miss the best choice, since it neglects the context.

To generate more accurate repair, we apply associate-rule-based method. It discovers association rules between a token candidate and its context and determines the repair value by such rules. Those methods will be elucidated and exemplified in Section 4.

3. REGEX-BASED STRUCTURAL REPAIR (RSR) ALGORITHM

For structural repair, we propose Regex-based Structural Repair Algorithm.

3.1 A Sketch of RSR algorithm

In this subsection, we give a formal definition of the structural repair problem and an overview of RSR algorithm. The target of structural repair is to convert the input string s to string s' which matches the given regex r and has the minimum edit distance to s. We define s' formally as follows.

DEFINITION 1. *Given a regex r and a string s, for a $s' \in L(r)$, if $\forall s'' \in L(r)$, $ed(s, s'') \geq ed(s, s')$, then s' is defined as the **best repair** to r of s, where $L(r)$ is the language of r and $ed(s_1, s_2)$ computes the edit distance between s_1 and s_2.*

The definition of *best repair* is suitable for describing the repair of s according to regex r. On one hand, s' matches r, so when s is transformed to s', it also matches r. Such transformation "repairs" s. On the other hand, since s' has the minimum distance to s, it represents a repairing method with the least cost. Therefore, the algorithm is to find the best repair of s to r.

We study the classical dynamic programming algorithm of computing edit distance between strings, which proves to be optimal [5], since the required algorithm involves it. In this algorithm, the prefixes of two input strings are used to form the subproblem [20]. Inspired by this, we attempt to track the distances of prefixes. However, '|' and closure in r make it impossible to directly use the prefix of r to form the subproblem.

Fortunately, since an NFA is used for matching strings to a regex, and a connected part of its states could match a substring, we could design a dynamic programming algorithm with a subproblem corresponding to a part of r's NFA and a prefix of s. We define the prefix of NFA.

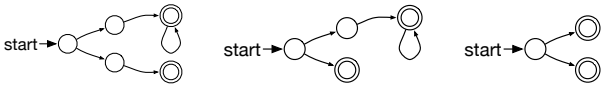
DEFINITION 2. *$L(A)$ denotes the set of all strings that could be accepted by an NFA A. s_p denotes a prefix of a string s which is one-character shorter than s. A **Prefix** of A, denoted by A_p, is an NFA that satisfies the following two conditions:*

1. *$\forall s \in L(A)$, $\exists A_p \in PS(A)$, s.t. $s_p \in L(A_p)$, where $PS(A)$ is the set of all prefixes of A.*
2. *The state transition diagram of A_p is a subgraph of that of A.*

We use Figure 1 illustrate NFA and its prefixes. From Definition 2, $\forall s_p \in L(A_p), \exists s \in L(A)$, length($s$)= length($s_p$)+1.

When a string s is accepted by an NFA A, the last character of s exactly matches some edges whose end states are A's accepting states. We call these edges *final edges*. The set of final edges of A is denoted by $TAIL(A)$.

Final edges can describe the relation between A and its prefixes. Given a string $s \in L(A)$, its prefix $s_p \in L(A_p)$, e and e_f denoting edges, and the function $tr(e)$ returns the transition symbol of e, $\exists e_f \in TAIL(A)$ that satisfies $s_p + tr(e_f) = s$.



(a) An NFA A. (b) Prefixes of NFA A.

Figure 1: An Example of Prefixes of NFA.

Structural repair with a string s and a regex in form of NFA N as input is defined as follows. Its subproblems are to find the edit distance between the prefixes of s and N.

DEFINITION 3. *Given a string s and an NFA N, the edit distance from s to N (denoted by $ed^*(s, N)$) is defined as the minimal edit distance between s and a string s' in $L(N)$. That is, $ed^*(s, N) = min\{ed(s, s')\}, s' \in L(N)$.*

Thus, given s_i as a prefix of s and an NFA A, the relationship between $ed^*(s_i, A)$ and its subproblems has four cases, where s_i denotes the prefix of s with length i. (1) s_i has been revised to match A_p and s_i is revised to match A by appending a token; (2) if s_{i-1} is revised to match A_p and the last token of s_i (denoted by $s[i]$) equals to $tr(e_f)$ for some $e_f \in TAIL(A)$, then s_i matches A; (3) if s_{i-1} is revised to match A_p and $s[i]$ equals to none of $tr(e_f)$, then s_i is revised to match A by substituting some $tr(e_f)$ for $s[i]$; (4) s_{i-1} has been revised to match A and s_i is revised to match A by deleting $s[i]$. In summary, this recursion function describes such relationship as follows.

$$ed^*(s_i, A) = min \begin{cases} ed^*(s_i, A_p) + 1 & (a) \\ ed^*(s_{i-1}, A_p) + f(s[i], e_f) & (b) \\ ed^*(s_{i-1}, A) + 1 & (c) \end{cases} \quad (1)$$

$f(s[i], e_f)$ is defined as follows.

$$f(s[i], e_f) = \begin{cases} 1 & s[i] \neq tr(e_f) \\ 0 & s[i] = tr(e_f) \end{cases}$$

The options marked (a), (b) and (c) correspond to different edit distance operations. If $c(A)$ denotes any character in $\{tr(e)|e \in TAIL(A)\}$, then (a) inserts $c(A)$ to the end of s_i; (b) substitutes $s[i]$ by a $c(A)$ when $f = 1$ or does nothing when $f = 0$ which means that $s[i] \in \{tr(e)|e \in TAIL(A)\}$; (c) deletes $s[i]$. These options are proposed to search the optimal solution of subproblems for the computation of $ed^*(s_i, A)$.

3.2 RSR algorithm

We discuss the details of RSR algorithm.

3.2.1 Data Structure

To store the intermediate results of the subproblems in Equation (1), we use a Matrix C with the minimum $C(i, e)$ representing $ed^*(s_i, A)$ for $e \in TAIL(A)$ and $s[i]$ matches e. That is, given S_e as the set of strings that pass e as the last edge, $C(i, e)$ is the minimum edit distance of $ed(s, s')$ for all $s' \in S_e$. Besides, for $e \in TAIL(A)$, $\bigcup S_e = L(A)(e \in TAIL(A))$.

Then, according to Equation (1), we have following recursion equation for C, where $PRE(e)$=$\{e'|e'.endstate = e.startstate\}$ and $e_p \in PRE(e)$.

$$C(i,e) = min \begin{cases} C(i,e_p)+1 & (a) \\ C(i-1,e_p)+f(s[i],tr(e)) & (b) \\ C(i-1,e)+1 & (c) \end{cases} \quad (2)$$

Since strings matching A must pass one of $e \in TAIL(A)$ and $C(i,e)$ records the minimum edit distance of strings through e, $\text{ed}^*(s_i, A)$ is computed from the collection of all possible $C(i,e)$. Likewise, with $TAIL(A_p) = \bigcup PRE(e)(e \in TAIL(A))$, $ed^*(s_i, A_p)$ is computed from the collection of $C(i,e_p)$ with $e_p \in PRE(e)$. As a result, Equation (2) is equivalent to Equation (1).

Besides C, we use another Matrix H to track the edit operators with items as triples (r,c,op), where r and c restore the repair sequence, and op records the operation. For each entry t in H, $t.r$ and $t.c$ is the row and column of the entry in C prior to t in the repair sequence, and $t.op$ is the repair operator with four choices, N, D, I and S. They are listed in Table 1. Column *Arg* shows the arguments of them. Note that only operator I and S have arguments.

Table 1: Arguments and Function of Operators

op	Arg	Function
I	token c	Insert c after s_r
S	token c	Substitute $s[r]$ by c
D	-	Delete the token at $s[r]$
N	-	Non-operation

3.2.2 Operator Selection Order

According to Equation (2), an operator is selected from four choices. Note that these operators are considered in different priorities in RSR Algorithm—for each row, D and S are considered in the first round and I is considered in the second round.

Such priority is caused by circles in NFA. Unlike the prefix relation of strings, the "previous" relation of edges in NFA characterized by $PRE(e)$ is not a strict partial order relation. For example, given edges a and b, if $a.startstate = b.endstate$ and $a.endstate = b.startstate$, $a \in PRE(b)$ and $b \in PRE(a)$, which violates the asymmetry of partial order. This unordered property results in uncertainty of $C(i,a)$ and $C(i,b)$ illustrated in Example 1.

To avoid uncertainty, deletions and substitutions are first considered, since these two operations involve only values in the $(i-1)$th row of C. However, insertions require the correctness of all $C(i,e_p)(e_p \in PRE(e))$ when calculating $C(i,e)$. After the first round, all $C(i,e)$'s in the ith row are temporally filled by deletions or substitutions. Then insertions are considered. If the cost of insertion is smaller for some $C(i,e)$'s, their op's are changed to I. Their r and c are also changed correspondingly.

The priority also solves the problem of the selection from multiple "best repair". The insertions are conducted secondly determining that given the equality of repair cost of a substitution or a deletion to an insertion, the insertion may not be considered. Hence, the string produced with the least edit distance is unique.

As the cost, the adjustment of the priority will decline the accuracy of the structural repair especially for regexes with a lot of small closures. The harm has two aspects, wrong deletion and hidden error.

Wrong deletion. Since Losses (repaired by insertions) are handled after Redundancies (repaired by deletions) and Displacements (repaired by substitutions), if a mistake deletion or substitution cost identically as the correct insertion, the accurate repair is missed. On the other hand, short token sequences such as bc in $a(bc)^*d$ lead to insufficiency of context information making the cost of the incorrect deletion no larger than that of the correct one.

Hidden error. It is caused by small closures in the regexes. Since the closure operator cannot count the number of tokens, the loss of a substring matching a single closure will not be discovered.

In conclusion, such inaccuracy comes from multi-repair-strategy problem. Note that similar cases in real occasions may hardly happen since few practical regexes have short token closures or disregard the number of tokens. More concrete and specific regexes can be used to solve this problem. For example, if strings that match ab^*c have more than 3 token b, then regex $abbbb^*c$ would be better than ab^*c. It can guarantee the number of b in repaired string is no less than 3. In addition, assigning a priority to tokens in the regex for multi-repair-strategy selection would contribute to the effectiveness of RSR algorithm. The design for this priority is left for future work.

3.2.3 Algorithm Description

The pseudo code of RSR algorithm is Algorithm 1. The inputs are a string s and an NFA A representing a regex $r(A)$ with E as the edge set and F as the set of accepting states. The outputs are the smallest edit distance between $r(A)$ and s and the edit operator sequence that converts s to match $r(A)$. We denote the number of edges in the shortest path from *start* to the end vertex of e as $dis(e)$.

Algorithm 1: RSR algorithm

Input: string s, NFA $A = (P, \Sigma, E, q_0, F)$.
Output: Minimal edit distance from s to A, generating Matrix H.

```
1: C(0,e) ← dis(e)_{e∈E}
2: C(i,φ) ← i_{i∈(0,len(s))}
3: for i ← 1 to len(s) do
4:    for all e ∈ E do
5:       if s[i] = tr(e) then
6:          C(i,e) ← min{C(i−1,e_p)_{e_p∈PRE(e)}}
7:          H(i,e) ← (i−1, e_p^m, [N])
8:       else
9:          C(i−1,e^m) ← min{C(i−1,e_p), C(i−1,e)}
10:         C(i,e) ← C(i−1,e^m) + 1
11:         if e^m ∈ PRE(e) then
12:            H(i,e) ← (i−1, e^m, [S, tr(e)])
13:         else
14:            H(i,e) ← (i−1, e^m, [D])
15:   Q ← E {Q is priority queue}
16:   while Q ≠ ∅ do
17:      e ←ExtractMin(Q)
18:      for all e_n ∈ NEXT(e) do
19:         if C(i,e) + 1 < C(i,e_n) then
20:            C(i,e_n) ← C(i,e) + 1
21:            H(i,e_n) ← (i, e, [I, tr(e_n)])
22: C(len(s), e_t^m) ← min{C(len(s), e_t)_{e_t∈TAIL(A)}}
23: return C(len(s), e_t^m) and H
```

Line 1 initializes each $C(i,\phi)$ by i since edit distances between strings and an empty NFA are i (i deletions) and each $C(0,e)$ by $dis(e)$ which means generating a string that reaches e with the least length.

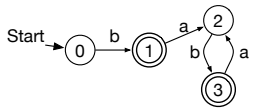


Figure 2: NFA of $b(ab)^*$

Figure 3: Position in Classical Algorithm

Table 2: Start

	ϕ	0/b	1/a	2/b	3/a
ε	**0**	**1**	**2**	**3**	**4**
b_1	**1**	0	2/1	2	4/3
b_2	**2**				
a	**3**				

Table 3: Final

	ϕ	0/b	1/a	2/b	3/a
ε	0	1	2	3	4
b_1	1	0	1	2	3
b_2	2	1	1	1	2
a	3	2	1	2	1

Table 4: Edges

Seq.#	s→e	tr	PRE	NEXT
0	$q_0 \to q_1$	b	ϕ	1
1	$q_1 \to q_2$	a	0	2
2	$q_2 \to q_3$	b	1,3	3
3	$q_3 \to q_2$	a	2	2

Line 3-21 compute $C(i,e)$ and $H(i,e)$. Due to different priorities, D and S operators are first considered in Line 4-14. Line 6 corresponds to Equation (2)-(b) with $f=0$, while Line 9 corresponds to (a) or (b) with $f=1$. Insertions are considered in Line 16-21, in which Line 19 judges whether insertion is better according to Equation (2)-(c).

Then Matrix C and H are derived. To obtain repaired string s_r and log s_{log}, Algorithm 2 shows a top-down traversal with H for input as s_r and s_{log} as output. Line 1 initializes variables for repair sequence generation in Line 3-6.

We clarify the distinction between RSR algorithm and edit distance computation as follows. Although experimental results state that edit distance calculation dominates the time cost ($> 99.5\%$), it is certain that merely edit distance computing cannot solve the entire problem. A procedure to generate the repaired string is needed. Furthermore, the generated token string on structure level is not enough. Hence, we propose "value repair" in Section 4. In summary, the distinction is computing edit distance is not the entire methodology for repair from structure level to value level.

We use a detailed example (Example 1) to elucidate the calculating process and a holistic one (composed by Example 2 and Example 4) to illustrate the overall repair.

Algorithm 2: String Restoring Method.

Input: String generating Matrix H, $len(s)$ and edge e_t^m derived in Algorithm 1.
Output: Repaired string s_r and logging s_{log}.
1: $s_r, s_{log} \leftarrow$ empty string
2: $\alpha \leftarrow \text{len}(s)$, $\beta \leftarrow e_t^m$
3: **while** $H(\alpha,\beta).r \neq 0$ **do**
4: $\quad s_r \leftarrow tr(H(\alpha,\beta).c) + s_r$
5: $\quad s_{log} \leftarrow H(\alpha,\beta).op + s_{log}$
6: $\quad \alpha \leftarrow H(\alpha,\beta).r$, $\beta \leftarrow H(\alpha,\beta).c$
7: **return** s_r and s_{log}

Example 1. *The inputs are $r = b(ab)^*$ and $s = bba$. The NFA of r is in Figure 2. The PRE and $NEXT$ set (if $e_p \in PRE(e)$, $e \in NEXT(e_p)$) and $tr()$ for edges are in Table 4, where a uniform number is assigned to each edge.*

The initialized entries are highlighted in bold in Table 2. We use subscripts to distinguish two b's in "bba". For the row of b_1, $C(b_1,e_i)$ is filled with only deletion or substitution in the first round as follows.

With $b_1 = tr(e_0)$ and $C(\varepsilon,\phi) = 0$, $C(b_1,e_0) = 0$; with $b_1 \neq tr(e_1)$ and $C(\varepsilon,e_0)+1 < C(\varepsilon,e_1)+1$, $C(b_1,e_1) = C(\varepsilon,e_0)+1 = 2$; with $b_1 = tr(e_2)$, $C(b_1,e_2) = C(\varepsilon,e_1) = 2$; with $b_1 \neq tr(e_3)$ and $C(\varepsilon,e_2)+1 < C(\varepsilon,e_3)+1$, $C(b_1,e_3) = C(\varepsilon,e_2)+1 = 4$.

Note that $C(b_1,e_1)$ and $C(b_1,e_3)$ are incorrect without insertion operations, which accounts for the "uncertainty" mentioned in Subsection 3.2. Thus we consider insertions in the follow steps.

With a higher priority of e_i implied by smaller $C(b_1,e_i)$, the priority queue $Q(b_1)$ is initialized as $\{e_0,e_1,e_2,e_3\}$. The following computation steps are as follows.

For e_0, with $e_1 \in NEXT(e_0)$ and $C(b_1,e_1) < C(b_1,e_0)+1$, $C(b_1,e_1) = 1$; for e_1, since $e_2 \in NEXT(e_1)$ and $C(b_1,e_2) = C(b_1,e_1)+1$, $C(b_1,e_2)$ is not modified; for e_2, $C(b_1,e_3)$ is changed to 3 for the same reason as $C(b_1,e_1)$; for e_3, $NEXT(e_3) = \emptyset$. Since Q is empty, the modification terminates.

The modified values in the row corresponding to b_1 are under the slash in Table 2. Executions for b_2 and a are similar to above procedure. Table 3 shows the final C. Since $TAIL(A) = \{e_0,e_2\}$, the minimal edit distance is $min\{C(a, e_0), C(a,e_2)\} = 2$.

We give an example about IDS rules about holistic repair.

Example 2. *Suppose a rule can be transformed to* `<Op><E1><SIP><SP><Op_dir><DIP><DP>[Opt_i:E2;Opt_j:v_j;]` *through lexical analyses.* `E1,E2` *are undiscernible tokens. The RSR algorithm runs as follows.*

For the interest of space, the detail of calculation is omitted. The edge list and transition symbols are in Table 6 and the derived Matrix C is given in Table 5, in which the path of the traversal is marked by underlines. Algorithm 2 generates s_r and s_{log}. E_1 and E_2 are substituted respectively by P and V_j.

$$s_r = \texttt{Op.P.SIP.SP.Op}_{dir}\texttt{.DIP.DP.[Opt}_i\texttt{:V}_i\texttt{;Opt}_j\texttt{:V}_j\texttt{;]}$$
$$s_{log} = [N][S, \texttt{<P>}]\underbrace{[N][N]\ldots[N]}_{12[N]}[S, \texttt{<V>}][N][N]$$

Table 5: Matrix C of Example 2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7	8	9	9	10	11	12	13	13
1	1	0	1	2	3	4	5	6	7	8	8	9	10	11	12	12
2	2	1	1	2	3	4	5	6	7	8	8	9	10	11	12	12
3	3	2	2	1	2	3	4	5	6	7	7	8	9	10	11	11
4	4	3	3	2	1	2	3	4	5	6	6	7	8	9	10	10
5	5	4	4	3	2	1	2	3	4	5	5	6	7	8	9	9
6	6	5	5	4	3	2	1	2	3	4	4	5	6	7	8	8
7	7	6	6	5	4	3	2	1	2	3	3	4	5	6	7	7
8	8	7	7	6	5	4	3	2	1	2	2	3	4	5	6	6
9	9	8	8	7	6	5	4	3	2	1	2	2	3	4	5	5
10	10	9	9	8	7	6	5	4	3	2	3	1	2	3	4	4
11	11	10	10	9	8	7	6	5	4	3	4	2	1	2	3	3
12	12	11	11	10	9	8	7	6	5	4	5	3	2	1	2	2
13	13	12	12	11	10	9	8	7	6	5	6	2	3	2	1	2
14	14	13	13	12	11	10	9	8	7	6	7	1	2	3	2	3
15	15	14	14	13	12	11	10	9	8	7	8	2	2	3	3	4
16	16	15	15	14	13	12	11	10	9	8	9	3	3	2	3	3
17	17	16	16	15	14	13	12	11	10	9	9	4	4	3	3	2

Table 6: Edges

#	e	tr(e)
1	0-1	Op
2	1-2	P
3	2-3	SIP
4	3-4	SP
5	4-5	Op_{dir}
6	5-6	DIP
7	6-7	DP
8	7-8	[
9	8-9	Opt
10	8-10	]
11	9-11	:
12	11-12	V
13	12-13	;
14	13-9	Opt
15	13-10	]

We could learn the difference between RSR algorithm and classical Levenshtein distance algorithm. In the classical approach, elements of the Matrix are ordered in the horizontal direction from left to right. When filling a cell, its left, above and left-above neighbors are involved, which constitutes a physical relation shown in Figure 3. The calculation of the grey cell needs the three cells pointed by arrows. As a comparison, in RSR algorithm, edges listed in neighbor columns may not be directly connected. The interconnections are maintained by PRE and $NEXT$. For example, in Row 13

and 14 of Table 5, the path is not connected physically but $edge_{14}$ is in $PRE(edge_{11})$.

3.3 Properties of RSR algorithm

This subsection shows the properties of RSR algorithm.

3.3.1 Correctness

THEOREM 1. *In Algorithm 1, $min_{e_t \in TAIL(A)}\{C(len(s), e_t)\}$ is the minimum edit distance between the string s and the regex r with NFA A.*

In order to prove Theorem 1, we first substantiate four lemmas.

LEMMA 1. *If e^* is the first edge extracted from priority queue Q in Line 17 of Algorithm 1 and $s' \in S_{e^*}$, then $C(i, e^*) = ed(s_i, s')$.*

Proof: Suppose that there exists $e_p^* \in PRE(e^*)$ with $C(i, e^*) > C(i, e_p^*) + 1$ satisfying the condition to modify $C(i, e_p^*)$ in Line 19. There must be $C(i, e_p^*) < min\{C(i, e)\} = C(i, e^*)$, which contradicts to the assumption that e^* is the minimum in Q. Thus, this lemma holds. ■

LEMMA 2. *After substitutions and deletions, for any $C(i, e) + 1 < C(i, e_n)_{e_n \in NEXT(e)}, C(i, e_n) = C(i, e) + 2$.*

Proof: Given edge a, b and e' where $b \in NEXT(a)$, $e' \in PRE(b)$ and

$$C(i, b) > C(i, a) + 1. \tag{3}$$

And according to Line 6 and Line 9, we have

$$C(i, b) = min\{C(i-1, e'), C(i-1, b)\} + 1 \tag{4}$$

$$\text{or } C(i, b) = min\{C(i-1, e')\}. \tag{5}$$

If (5) holds, then

$$C(i, b) \leq C(i-1, a) \tag{6}$$

Considering the comparison between $C(i-1, a)$ and $C(i, a)$. It is possible that

$$\begin{cases} C(i-1, a) = C(i, a) - 1 \\ C(i-1, a) = C(i, a) \\ C(i-1, a) = C(i, a) + 1 \end{cases}$$

If $C(i-1, a) = C(i, a) + 2$ holds, then only 1 step, cutting $s[i]$, is required to make $C(i-1, a) = C(i, a) + 1$. Hence,

$$C(i-1, a) \leq C(i, a) + 1. \tag{7}$$

Combining with (6), we get $C(i, b) \leq C(i, a) + 1$, which contradicts to (3). Hence, $C(i, b) = min\{C(i-1, e'), C(i-1, b)\} + 1$.

On the other hand, $min\{C(i-1, e')\} \geq min\{C(i-1, e'), C(i-1, e)\}$. Therefore, $C(i, b) \leq min\{C(i-1, e')\} + 1 \leq C(i-1, a) + 1$

Combining with (7), it holds that

$$C(i, b) \leq C(i-1, a) + 1 \leq (C(i, a) + 1) + 1 = C(i, a) + 2 \tag{8}$$

Due to (3) and (8): $C(i, b) = C(i, a) + 2$. ■

LEMMA 3. *For each e extracted from Q, $C(i, e)$ is stable and optimal, that is, for $s' \in S_e$, $C(i, e) = ed(s, s')$.*

Proof: The only possible case invalidates this lemma is that when an edge e is pop from the Q, $C(i, e)$ is minimal. In such case, consider edges e and e_k, which is inferior to e in Q, with $e \in NEXT(e_k)$. When an edge in $PRE(e)$ is extracted and modifies $C(i, e_k)$, $C(i, e)$ could possibly be lessen as well for $e \in NEXT(e_k)$. Since e is extracted before e_k, $C(i, e_k)$ will not "correct" $C(i, e)$, as will result in the incorrectness of $C(i, e)$.

We now prove that above case would never happen. Supposing that $C(i, e_k)$ is modified from x to $x-1$, with $C(i, e) \leq x$, the new $C(i, e_k)$ has only two possibilities, $C(i, e_k) = C(i, e) - 1$ or $C(i, e_k) \geq C(i, e)$. Above case happens only under the former condition where $C(i, e_k) = C(i, e) - 1$. However, according to Lemma 2, if $C(i, e)$ is about to modify $C(i, e_k)$, $C(i, e) = C(i, e_k) + 2$. Therefore, we have proved that for any e in E, $C(i, e)$ will not be modified by edges with smaller key in Q. That is, for all e extracted from Q, $C(i, e)$ is stable and optimal. ■

LEMMA 4. *Given an NFA A, A's prefix A_p and edge $e \in TAIL(A_p)$, if for any A_p, $C(i-1, e)$'s cover $ed^*(s_{i-1}, A_p)$, then $C(i, e)$ cover any $ed^*(s_i, A_p)$ after Algorithm 1.*

Proof: In Subsection 3.1 and Subsection 3.2, we proposed the recursion function (1) and its equivalent form (2) expressed by C. Given this equation, if we substantiate that $C(i, e)$ in Algorithm 1 is filled strictly according to (2), then the correctness of Lemma 4 is proven. On the other hand, we reform the execution order of the recursion function by the priority of operations mentioned in Subsection 3.2. Its correctness can be proven by the combination of Lemma 1 and Lemma 3. Since for the first and the following edges e extracted from Q, $C(i, e)$ is the optimal. Therefore, Lemma 4 is validated by the correctness of (2) and the reformation above. ■

Then, we give the proof of Theorem 1.

Proof(Theorem 1): Since this algorithm is based on dynamic programming, its correctness is proven by inductive method. We apply inductions on i, the length of prefixes of s. When $i = 0$, all $C(\varepsilon, e)$ with $e \in E$ are initialized as $dis(e)$. Because any edit distance from an empty string ε to a string passes $e \in TAIL(A)$ in $L(A)$ is exactly $dis(e)$, $C(\varepsilon, e)$'s contain $ed(\varepsilon, A)$.

The inductive hypothesis is that all $C(i-1, e)$ are correct for each e and they contain optimal distance values of any $ed^*(s_{i-1}, A)$ given $e \in TAIL(A)$. We attempt to prove that $C(i, e)$'s can then provide valid $ed^*(s_i, A)$ under above hypothesis. Lemma 4 proves the inductive hypothesis. Therefore, $C(len(s), e)$ records $ed^*(s, A_p)$ with $e \in TAIL(A_p)$. Particularly when $e_t \in TAIL(A)$, $C(len(s), e_t)$ gives edit distances from s to strings in $L(A)$. $ed^*(s, A) = min\{C(len(s), e_t)\}$, since it has the minimum repairing cost. Theorem 1 is proven. ■

3.3.2 Complexity

The complexity of Algorithm 1, shown in Theorem 2, is determined by the length of string s (denoted by n) and the number of edges in NFA A (denoted by m).

THEOREM 2. *The time complexity of RSR algorithm is $O(nm^2)$ and the space complexity is $O(nm)$.*

Proof: RSR algorithm's time complexity is comprised of three parts: the cost of initialization (denoted by T_1), the

cost of computation of C and H (T_2) and the cost of repair sequence generation (T_3). Clearly, $T = T_1 + T_2 + T_3$.

The time complexity of the initialization (Line 1 in Algorithm 1) is $T_1 = O(m) + O(n)$.

During the computation of C and H, 'for' loop beginning at Line 3 will be executed for n times. Therefore, $T_2 = n \times (T_{2.1} + T_{2.2} + T_{2.3})$, where $T_{2.1}$, $T_{2.2}$ and $T_{2.3}$ correspond to the time complexities of 'for' loop (Line 4-14), creating priority queue (Line 15) and 'while' loop (Line 16-21), respectively. In the worst case, as for Line 6 and Line 9, $|PRE(e)| = m$. Thus the cost of traversing $PRE(e)$ to get the minimum $C(i, e_p)$ is m and $T_{2.1} = O(m \times m) = O(m^2)$.

Time complexity of initializing the priority queue Q is $T_{2.2} = O(m \lg m)$. In the 'while' loop at Line 16, the number of edge extraction is m and in the worst case, $|NEXT(e)| = m$. Thus, $T_{2.3} = O(m \times m) = O(m^2)$. Therefore, $T_2 = n \times (O(m^2) + O(m \lg m) + O(m^2)) = O(nm^2)$.

The cost of finding $min\{C(len(s), e_t]\}$ is $|TAIL(A)|$, which equals to m in the worst case. It takes at most $max\{len(s) + min(dis\ (TAIL(A)))\}$ steps to restore the string in H. Thus, the maximum $min(dis(TAIL(A))) = m$. It implies that $T_3 = O(m) + O(max\{n + m\}) = O(max\{m + n\})$.

In summary, the entire time complexity: $T = T_1 + T_2 + T_3 = O(m) + O(n) + O(nm^2) + O(max\{m + n\}) = O(nm^2)$

For space complexity, both Matrix C and H have $n + 1$ rows and $m + 1$ columns, thus the space complexity is $S = O(2 \times (m + 1) \times (n + 1)) = O(mn)$. ■

Note that regexes are not very long in practice. Thus, m is usually far less than n and the time and space complexity are approximately linear with n.

3.4 Optimization for RSR algorithm

Since time complexity of computing C dominates the time cost, we attempt to reduce the time for this step.

In Algorithm 1, all $e \in E$ are traversed to get $C(i, e)$ in the ith iteration. However, when $dis(e)$ is far larger than i, the computation of $C(i, e)$ may be redundant because many insertions are involved to generate for repair. Since such insertions are meaningless, we attempt to reduce the computation cost by avoiding them. Thus, we compute $C(i, e)$ just for i large enough. That is, $C(i, e)$ is computed only when $dis(e) \leq i + 1$. It means in the ith iteration, only edges e with $dis(e) \in \{1, 2, \ldots, i, i + 1\}$ are calculated.

To ensure that such optimization does not cause illegal computation, it could be applied when $len(s) \geq max\{dis(e), e \in TAIL(A)\}$. It is because an $e_t \in TAIL(A)$ with $len(s) < dis(e_t)$ implies $C(i, e_t) = \infty$ when the optimization is applied and then this will clearly lead to an illegal computation in the following steps.

According to above discussion, the optimized algorithm is in Algorithm 3. Since the optimization focuses on C, Algorithm 3 shows only the computation of C. The computation of H and repair generation are unchanged.

Compared with Algorithm 1, Algorithm 3 conducts the optimization in two positions. One is Line 6 ensures that $C(i, e)$ is computed only when $dis(e) \leq i + 1$. The other is Line 11, which avoids $C(i, e) = \infty$ to be inserted into Q. We use an example to illustrate it.

EXAMPLE 3. *The input is $r = ab(c|d)e^*fg$ and $s = bxelg$. $length(s) = 5$, $max\{dis(e)\} = 5$ which are very close. The result Matrix C is in Table 7. The computation cost of blank cells is saved. In this problem, 19/55 time cost of Line 5-16 is saved according to Table 8.*

Algorithm 3: Optimized Algorithm

Input: String s, NFA $A = (P, \Sigma, E, q_0, F)$
Output: Optimized edit distance s to A

```
1: for all e ∈ E do
2:     C(0, e) ← dis(e)
3: for i ← 0 to len(s) do
4:     C(i, φ) ← i
5: for i ← 1 to len(s) do
6:     for all e ∈ E with dis(e) ≤ i + 1 do
7:         if s[i] = tr(e) then
8:             C(i, e) ← min{C(i − 1, e_p)}_{e_p ∈ PRE(e)}
9:         else
10:            C(i, e) ← min{C(i − 1, e_p), C(i − 1, e)} + 1
11:    Q ← E' {E' ← {e|dis(e) ≤ i + 1, e ∈ E}}
12:    while Q ≠ ∅ do
13:        e ←ExtractMin(Q)
14:        for all e_n ∈ NEXT(e) do
15:            if C(i, e) + 1 < C(i, e_n) then
16:                C(i, e_n) ← C(i, e) + 1
17: return min{C(len(s), e_t)}, (e_t ∈ TAIL(A))
```

Table 7: Optimized Matrix C

Seq.#	ϕ	0	1	2	3	4	5	6	7	8	9	10
ϕ	0	1	2	3	3	4	4	4	4	5	5	5
b	1	1	1									
x	2	2	2	2	2							
e	3	3	3	3	3	2	3	2	3			
l	4	4	4	4	4	3	4	3	4	3	3	4
g	5	5	5	5	5	4	5	4	5	4	4	3

Table 8: Cost

i	$s[i]$	dis_{max}	Saved
0	b	1	9
1	x	2	7
2	e	3	3
3	l	4	0
4	g	5	0

Time complexity of the optimization is $O(nm^2)$ and space complexity is $O(nm)$ with m denoting the number of edges in NFA A and n denoting the length of string s. Since this strategy prunes some calculation steps, it speeds up RSR algorithm although their time complexities are identical.

As a trade-off, the optimization sacrifices some accuracy—the results of it are not always optimal because the "redundant information" eliminated may contain clues of optimal solutions. Fortunately, such cases are rare because the possibility of eliminating essential information would decrease as length of the prefix of s grows. When $i > max\{dis(e)_{e \in TAIL(A)}\} - 1$, no essential information will be lost since the number of pruned calculation steps gets 0.

4. VALUE REPAIR

The goal of value repair is to find a proper value from the available option values for the variables of I and S operators in the repair sequence. Given $V(t)$ as the value set of token t, `<P>` as the protocol token of Snort Rules and $V(P)$= {`tcp,udp,ip,icmp`}, the value repair answers the following questions.

- What should be inserted with operator `[I,<P>]`? Which option in $V(P)$ is the most suitable one?
- If an erroneous token written as `txp` should be substituted by a `<P>` with operator `[S,<P>]`, how to make the choice? What if the wrong token is written as `xxx`?

We propose two approaches respectively based on edit distance and associate rules. These approaches are only concerned about repairing tokens with finite and closed value sets. For the instances that we have known, it is common for the tokens to have finite and closed value sets. The `<Protocol>` token has four options: {`tcp, udp, ip, icmp`} and token `<Operation Direction>` has two: {<>,->}. In

the "MovieLens" sets, `<Movie Type>` token has 20 options such as "Mystery", "Sci-Fi", ..., etc., which is also finite. As for the consecutive and continuous values cases, we discretize them into finite sets before utilizing those two approaches.

4.1 Edit-distance-based Method

Since edit distance is effective to measure the similarity of strings, it is natural to select the value with the smallest edit distance to the original string. For example, it is reasonable to repair `txp` to `tcp`, since it has the minimal edit distance to `txp` in $V(P)$.

To implement such method efficiently, we use an efficient top-1 string similarity search algorithm [8] on the option set.

The advantage is this method is easy to comprehend and implement without requiring extra knowledge of data. It can be pretty efficient. The disadvantage is when the operator is insertion and the token is completely wrong-written, this method can hardly make convincing choices without sufficient clues. To make rational choices in these cases, we give an association rules (AR) based method for supplement.

4.2 AR-based Method

Association rules mining aims to find the item sets that co-occur frequently. For our problem, it is effective to find the co-occurrence relationship between one value and its context. Then, the true value could be implied.

We use this example to illustrate this method. For the Snort Rules, in each entry with both `<Source Port>`= 80 and `<Destination Port>`= 80, the `<Protocol>` must be `<tcp>`, because 80 is the port number of tcp services. Hence, we can claim if string s mistakenly writes `<P>` as `xxx` with 80 as the values of source and destination port numbers, then a correct repair is to modify `xxx` to `tcp`—{`<Source Port>`= 80,`<Destination Port>`= 80} implies {`<P>`=`tcp`}. This is regarded as an association rule. It describes the extent of association or dependency between the two sets.

Formally, an AR is defined as an implication in form of $X \Rightarrow Y$, where X and Y are item sets and $X \cap Y = \emptyset$ [2].

The example of Port and Protocol can be explained by an AR. In the Port-Protocol Rule $R_{p\dot{p}}$:$X \Rightarrow Y$, X is {`<S.Port>=80`, `<D.Port>=80`} and Y is {`<P>=tcp`}.

For an AR, *confidence* and *support* are often used to evaluate its usability. Given $count(X)$ as the number of transactions containing X, $count(X, Y)$ as the number of transactions containing both X and Y, and $count(T)$ as the size of the entire transaction set, $support(X) = \frac{count(X)}{count(T)}$, $support(X, Y) = \frac{count(X,Y)}{count(T)}$. *confidence* measures the confidence level of a rule. Given a rule $R : X \Rightarrow Y$ with X, Y as item sets, *confidence(R)* is: $confidence(R) = \frac{support(X,Y)}{support(X)}$.

In our problem, the association rules have two extra constraints compared with the general ones. One is that tokens with infinite or continuous values should be discretized into finite and closed sets before using these methods. The other is that only rules whose size of consequent set (Y) are 1 are included in association rule set, for structural repair is applied on single token. Thus only the rules with a single token as right side are required. With these constraints, the associate rules used in our approach could be mined with Apriori algorithm [3] directly.

After rule mining of the set, an AR set Γ is derived. Each rule in Γ is in form of $R : X \Rightarrow y$. Given the value set of token t as $V(t)$ and $\Gamma_t = \{R | R \in \Gamma, y \in V(t)\}$, Γ_t contains all association rules related to t. Therefore, for strings containing tokens with values in the X of R, we can use R for the repair and y is selected to repair t.

Such method takes advantage of the context or the association relations to determine the values. As a tradeoff, it requires extra efforts for association-rule discovery.

4.3 Value Repair Method Selection

As discussed before, these two methods are complementary. In this subsection, we discuss how to choose a proper value repair method.

We first define comparable measures for the approaches and select by comparing these measures. As we know, for an association rule $R : X \Rightarrow y$, $confidence(R)$ describes the possibility of correctness and the reliability of amendment of token t using X. For methods comparison, we propose "confidence" of edit-distance-based method denoted by γ as $\gamma = 1 - \frac{min\{ed(s,V(t))\}}{len(s)}$, where s is the wrongly-written string. Since γ is the largest proportion of correct characters in s, the closer s is to some string in $V(t)$, the more possible this repair is correct. Therefore, γ could perfectly describe the confidence of the edit-distance-based repair.

With the definitions of confidences, we propose the selection strategy. For substitution operator, the strategy with maximum confidence is selected. Since insertion operator has no original value as reference, the value repair depends on the association rules. The strategy is as follows.

Given a token t, its original value s, $confidence(R_t)$ and γ_t, for a substitution operator `[S,t]`, judge whether R_t exists such that $confidence(R_t) > \gamma_t$. If so, t is repaired according to the R_t with maximal $confidence(R_t)$. Otherwise, t is set to s' in $V(t)$ with minimal $ed(s, s')$. For an insertion operation `[I,t]`, find the rule $R : X \Rightarrow y$ with maximum $confidence(R_t)$ and t is set to y. The following example continues Example 2 to explain the value repair.

EXAMPLE 4. *Suppose E_1 is written as "itcmp" while the value set of P is {`tcp,ip,icmp,udp`}, `<SIP> <DIP>` are both 80, and $R : \{SIP = 80, DIP = 80\} \Rightarrow \{P = tcp\}$ is an AR mined from data set, $confidence(R) = 0.9$, the comparison of the two methods is as follows.*

According to the edit-distance based method, the confidences of values of $V(P)$ are as follows, $\gamma_{ip} = 0.4$, $\gamma_{tcp} = 0.6$, $\gamma_{icmp} = 0.8$, $\gamma_{udp} = 0.2$. Transforming E_1 to "icmp" is a wise choice. But according to the AR-based method, rule R suggest that E_1 be modified to "tcp". Since $confidence(R) > \gamma_{(icmp)}$, we select "tcp" for repair.

5. EXPERIMENTS

In this section, we show the results and analyses of the effectiveness and efficiency.

5.1 Experimental Settings

Environment. We implement the algorithms by C++ compiled by GCC on a PC with an Intel Core i7 2.1GHz CPU, 8GB memory, and 64bit Windows 8.1.

Data Sets. To test our algorithm comprehensively, we use both real and synthetic data. For real data, we choose the intrusion detection rules of Snort[1] and MovieLens user

[1]Available at https://www.snort.org/downloads/community/community-rules.tar.gz. It contains rules that were created by Sourcefire, Inc.

rating data[2]. As is mentioned, IDS uses "rules" to record the properties of malicious Internet packets. These rules are written in an language that can match a regex. MovieLens rating data is collected by GroupLens Research from the MovieLens web site [1]. Its regex pattern is in Section 1.

For synthetic data, the verification of shortcomings about shorter and more closures and options mentioned in Subsection 3.2 should be considered. To achieve this, we artificially create 4 regexes by character `a-z` with different traits. Particularly, Regex2 and Regex4 have more and shorter closures than Regex1 and Regex3, which are

Regex 1	a*b((cd)|(de)|(ec))*f((gh)|(ij)|(klm))*n((o(pqz)*)|((r|s)t*))((uv)|(wxy))*
Regex 2	(ab)*cde*(fgh)*((ij)*kl)*m(n*op)*((qrs)*(tu)*)*(vwxy)*z
Regex 3	((ab)|c)*((def)|(gh))*(ijk)*((lmn)*|(opq)*)*((rst)|(uvw)|(xyz))*
Regex 4	((ab)|(cd))*ef((ghi)*|(jk)*)((lm)*|(nop)*)*((qr)*s|t(uv)*)*w(x|y)*z

To control the error ratio, we inject errors to the strings. For each string in the data set, we choose μ proportion of positions and randomly revise, delete the token or insert a token in such position. Meanwhile, we keep a copy of the uncontaminated data and exploit it as the ground truth.

We consider 4 parameters that may affect the performance: the number of strings in data set, denoted by M; the average proportion of error characters in the strings, denoted by μ; the average length of data strings, denoted by $\bar{L}_s$; the error type, denoted by T. The default settings are $M = 2,000$, $\mu = 5\%$, $\bar{L}_s = 45$, and $T = \{L, R, D\}$ (representing Loss, Redundance and Displacement).

Measures. We considered to use Precision and Recall with following definitions to measure the performance.

$$Precision = \frac{|incorrect\ entries| \bigcap |actual\ repairs|}{|actual\ repairs|}$$

$$Recall = \frac{|incorrect\ entries| \bigcap |actual\ repairs|}{|incorrect\ entries|}.$$

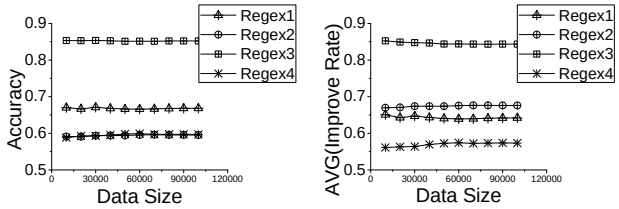
Incorrect entries is the set of erroneous items in data sets artificially injected and *actual repairs* is the set of amendments actually conducted. In the experimental results, we find Precision and Recall are always equal to 1. The reason is all error tokens can be discovered due to disobedience to the regex and then repaired, which means that *incorrect entries* = *actual repairs*. To test the effectiveness of the algorithms more deeply, we use Accuracy and Improve Rate as the measures.

Before explaining the parameters, we define set AC and W. AC is the set of RSR algorithm repaired strings (each one is uniquely produced) that are equal to the ground truth. W is the set of strings with injected errors. The *Accuracy*, denoted by λ, is defined as $\lambda = \frac{|AC|}{|W|}$. It represents the overall repair performance. That is, RSR algorithm's capability to restore the entire data set.

We also use *Improve Rate* ω to describe the performance regarding individual data items and the extent to which a single item is repaired. Given the original string s_o, error string s_e, and the repaired string s_r, $\omega = 1 - \frac{ed(s_o,s_r)}{ed(s_o,s_e)}$. We use $\bar{\omega} = \frac{\Sigma_{i=1}^{|W|}\omega_i}{|W|}$ to show the overall effectiveness. Notice that $\bar{\omega} < 0$ is means a repair with negative impact.

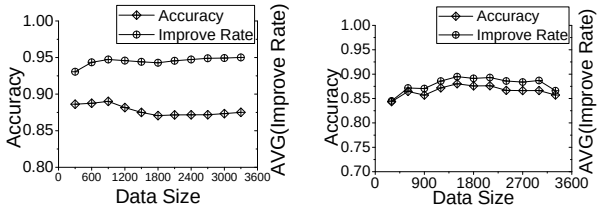
To measure efficiency, we use the time duration of execution of the algorithm, denoted by t. The unit of run time is second (s).

[2]Available at http://grouplens.org/datasets/MovieLens



(a) M vs. λ (b) M vs. $\bar{\omega}$

Figure 4: The Impact of Data Size M for Synthetic Sets



(a) Snort Rules (b) MovieLens Rating

Figure 5: The Impact of Data Size M for Real Sets

5.2 Results and Analysis

In this subsection, we show experimental results and analyses of λ and ω when varying M, μ, $\bar{L}_s$ and T.

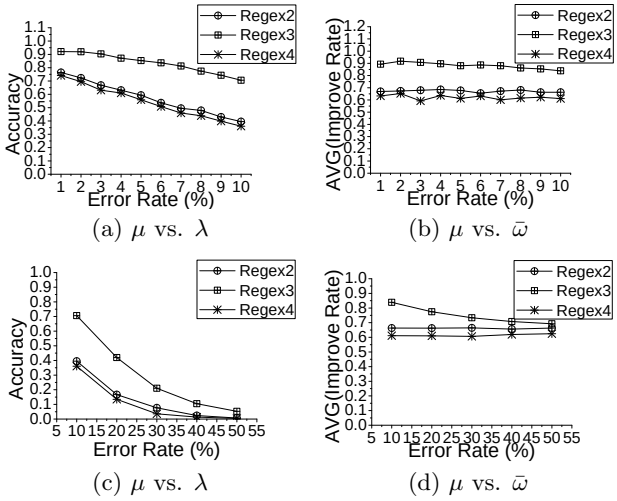
5.2.1 Experiments on Effectiveness

The Impact of Data Size (M). To test the RSR algorithm's scalability, we execute it on 40 synthesis sets of 4 regexes with M varying from 10,000 to 100,000, 22 real sets of the Snort Rules and the MovieLens ratings with M varying from 300 to 3,300. Experimental results are shown in Figure 4 and Figure 5. These figures manifest that M does not affect the accuracy and the improve rate. The reason is that sequence repair only concerns about individual items rather than the overall data set.

Besides, the performances vary significantly among regexes. In Figure 4, the accuracy and the average improve rate of Regex 3 are larger than those of Regex 2 and Regex 4 due to the disadvantage of the priority of RSR algorithm. As a comparison, the regexes of the Snort Rules and MovieLens ratings with longer closure sequences achieve better λ and $\bar{\omega}$ than those on synthetic sets, as shown in Figure 5.

The Impact of Error Rate (μ). To evaluate the impact of μ, we assign μ in {1%-10%} and {10%-50%}. Since data for Regex 1 with average length 50 is not enough due to the shortage of closures, we abandon such data sets.

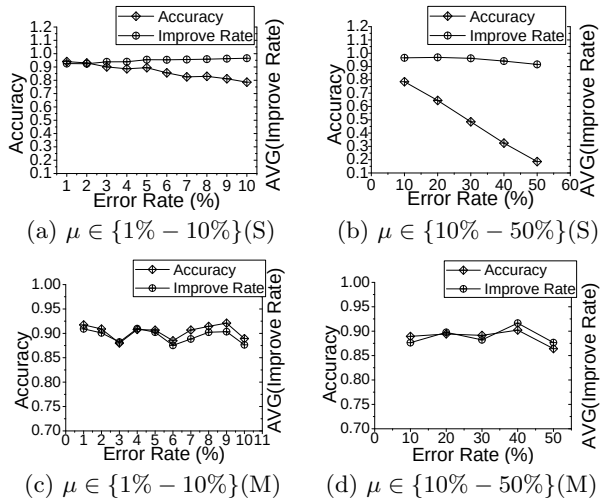
The experimental results on synthetic and real data are shown in Figure 6 and Figure 7 respectively. From the results, we observe that the accuracy decreases along with the growth of μ. Such phenomenon is explained as follows. Suppose the probability that an error is correctly repaired is p_a and the string s has $q = \mu \cdot len(s)$ errors, then λ of this set is $\lambda = p_a^q = p_a^{\mu \cdot len(a)}$, which means that λ is exponential to μ and would decrease while μ increases. Accuracies of Figure 7(c) and Figure 7(d) decrease slowly with fluctuations since the structural context information of the regex of MovieLens data set is sufficient, which contribute to accuracy and enlarges p_a to make it close to 1.



(a) μ vs. λ (b) μ vs. $\bar{\omega}$

(c) μ vs. λ (d) μ vs. $\bar{\omega}$

Figure 6: The Impact of Error Rate μ for Synthetic Sets



(a) $\mu \in \{1\% - 10\%\}$(S) (b) $\mu \in \{10\% - 50\%\}$(S)

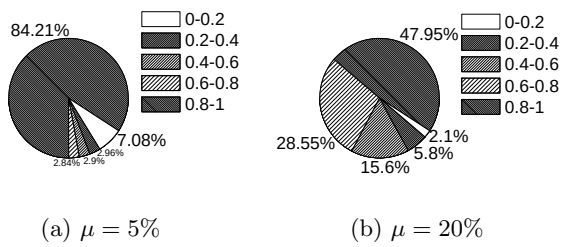
(c) $\mu \in \{1\% - 10\%\}$(M) (d) $\mu \in \{10\% - 50\%\}$(M)

Figure 7: The Impact of Error Rate μ for Real Data Set

According to Figure 6(b), Figure 6(d) and Figure 7, $\bar{\omega}$ keep around unchanged on both sets showing an stable repairing effectiveness of RSR algorithm. We show two pie-charts of the proportion of ω in data set of Regex 3 in Figure 8(a) with $\mu = 5\%$ and Figure 8(b) with $\mu = 20\%$ to observe its distribution. From these figures, most of individual improve rates are in $[0.8, 1]$ and the items with $\omega > 0.6$ is over 75%. It shows that our approach is effective.

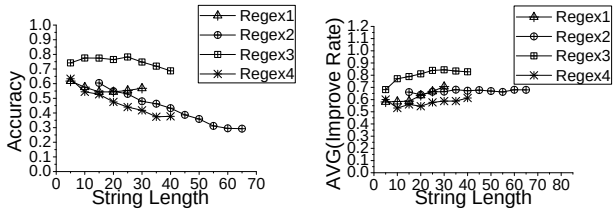
The Impact of Average Length ($\bar{L}_s$). Since the length of real data in uncontrollable, we only conduct experiments on the synthetic data sets divided by string length. Group 1 has strings with length in $[1, 5)$, Group 2 in $[5, 10)$, Group 3 in $[10, 15)$, and so on. Only groups with over 2,000 items are considered in this experiments. The results are Figure 9. The accuracy decreases when strings get longer, which is also caused by $\lambda = p_a^q = p_a^{\mu \cdot len(a)}$. Additionally, $\bar{\omega}$ is still stable.

To illustrate the distribution of ω of Regex 3, we also draw two pie-charts for $\bar{L}_s = 15$ and $\bar{L}_s = 65$ in Figure 10. Similar to Figure 8, high improvement repairs dominate the cases



(a) $\mu = 5\%$ (b) $\mu = 20\%$

Figure 8: Proportion of Improve Rate ω for Regex 3



(a) $\bar{L}_s$ vs. λ (b) $\bar{L}_s$ vs. $\bar{\omega}$

Figure 9: The Impact of Average String Length $\bar{L}_s$ for Synthetic Data Sets

which also shows the effectiveness.

The Impact of Error Type (T). To test the impact of error types, we inject only one error type T for each group with the same μ. The results of synthetic and real data are in Table 9. We observe that when $T = L$ (Loss), the λ and $\bar{\omega}$ of Regex 2 and Regex 4 are much lower than that of Regex 3. This is another evidence of disadvantages of priority mechanism of RSR algorithm. Besides, λ and $\bar{\omega}$ about Redundance and Displacement and the overall performance on the real data are all higher than 0.7.

Table 9: The Impact of Error Type T on Performance

Error Type (T)	R		L		D	
Performance	λ	$\bar{\omega}$	λ	$\bar{\omega}$	λ	$\bar{\omega}$
Regex 2	0.82	0.91	0.23	0.13	0.88	0.94
Regex 3	0.98	0.98	0.57	0.58	0.94	0.97
Regex 4	0.74	0.86	0.22	0.05	0.82	0.90
Snort Rules	0.95	0.97	0.88	0.92	0.93	0.93
MovieLens Rating	1	1	0.85	0.84	0.91	0.91

5.2.2 *Experiments on Efficiency*

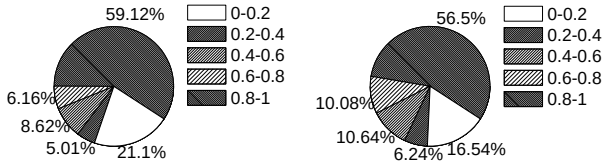
Since the efficiency is affected by the data size and string length, we test the impact of these two parameters.

We vary the data size M from 10,000 to 100,000 in synthetic sets and from 300 to 3,300 in the real sets. The results are in Figure 11. From such results, it is observed that the run time t is around linear with M. The differences of slopes are caused by the difference of the average length of the four sets and the difference of edge numbers of the four regexes' NFA which dominate the size of Matrix C.

We varies the string length from 5 to 70. The relation of $\bar{L}_s$ and run time t is shown in Figure 12. t grows along with L_s around linearly, which coincides the time complexity $O(nm^2)$. μ does not affect the execution time. Table 10 shows the relationship between μ and t. From the results,

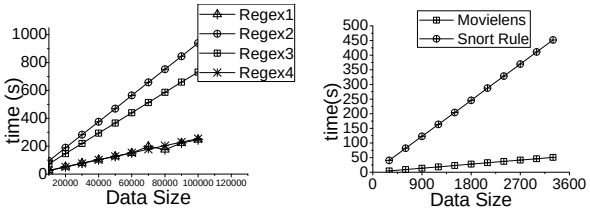
Table 10: The Impact of Error Rate μ on Run Time

μ	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%	20%	30%	40%	50%
Regex 2	3.57	3.60	3.59	3.60	3.63	3.54	3.60	3.62	3.56	3.57	3.62	3.54	3.54	3.50
Regex 3	4.63	4.66	4.62	4.79	4.69	4.79	4.70	4.74	4.71	4.72	4.66	4.74	4.70	4.51
Regex 4	4.58	4.53	4.62	4.60	4.70	4.66	4.64	4.67	4.53	4.58	4.44	4.46	4.47	4.38



(a) $\bar{L}_s = 15$ (b) $\bar{L}_s = 65$

Figure 10: Proportion of Improve Rate ω for Regex 3



(a) Synthetic Data Sets (b) Real Data Sets

Figure 11: The Impact of Data Size M on Run Time

our algorithm repairs 2,000 strings with average length 45 within 5s, which shows the efficiency of it.

The relation of T and t is given in Table 11. Redundant errors cost slightly less than Losses and Displacements, since deletions are considered in the first round that lessen the calculation pressure of the second one.

5.2.3 Experiments on Optimization

We evaluate the acceleration impact of the optimization strategy on both synthetic and real data sets and all parameters are under default setting. We compare run time, Accuracy λ and Improve Rate $\bar{\omega}$ of the unoptimized algorithm (Algorithm 1) with the optimized one (Algorithm 3) to observe the tradeoff of correctness and efficiency.

Table 12 shows the comparison between the run time of the original and optimized algorithm for synthetic data. λ

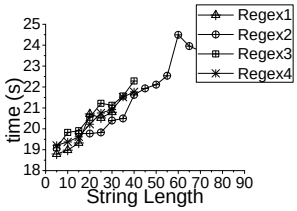


Figure 12: The Impact of Average String Length $\bar{L}_s$ on Run Time

Table 11: The Impact of Error Type T on Run Time

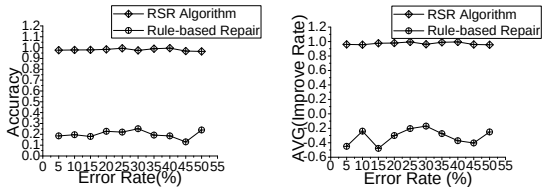
T	L	R	D
Regex 2	3.80	3.46	3,62
Regex 3	4.89	4.42	5.10
Regex 4	4.65	4.37	4.53
Snort Rule	134	133.50	133.50
MovieLens Rating	31.19	31.07	31.08

Table 12: Optimization for Synthetic Data Sets

Regex	1	2	3	4
Original	11.04	10.02	13.42	13.25
Optimized	10.18	9.12	13.05	12.08

Table 13: Optimization for Real Data Sets

Data Set	Algorithm	t	λ	$\bar{\omega}$
Snort Rules	Original	495.80	0.872	0.945
	Optimized	411.03	0.854	0.920
MovieLens Ratings	Original	18.205	0.876	0.891
	Optimized	10.32	0.846	0.821



(a) Comparison on λ (b) Comparison on $\bar{\omega}$

Figure 13: Comparison on Different Repair Methods

and $\bar{\omega}$ of these methods are identical. Table 13 gives the results for real data. The evaluation shows that the optimization strategy has considerable capability on cutting down the time cost (an average of 7.1% for synthetic data and 30.2% for real data), especially in long data strings and pattern regex. On the other hand, it can still maintain the effectiveness on a high level since the average loss of accuracy in real data is less than 3%.

5.2.4 Comparison with Dependency-Rule-Only Repair

In order to compare difference of effectiveness of structural repair between RSR algorithm and Dependency-Rule-Only repair strategy, we derive the dependencies from the structure requirements and measure the λ and $\bar{\omega}$ on various amount of errors by controlling μ. For example, if the regex is $xA|yB$, then we derive that $x \to A$ and $y \to B$. Figure 13 shows the results.

For Accuracy (λ) in Figure 13(a), RSR algorithm is higher than Dependency-Rule-Only method. For Improve Rate ($\bar{\omega}$) in Figure 13(b), the Dependency-Rule-Only method shows negative impact, indicating that it does not suit for structural repair.

5.2.5 Summary

In summary, we can draw following conclusions from the experiments.

- Compared with dependencies, regex-based method is more applicable to structural repairs, especially for regex constrained data. Dependencies, due to lack of structural insight, performs poorly and even harmfully on this repairing topic.
- Data size M has no effect on accuracy and improve rate. The accuracy λ decreases along with the growth of μ and L_s which can be explained by $\lambda = p_a^{\mu \cdot L_s}$. However, the average improve rate $\bar{\omega}$ would remain stable regardless of μ or L_s. As for error type T, losses are more difficult to discover and repair due to the priority in RSR algorithm, especially for errors in small closures. In all, the proposed algorithm could repair the errors according to the regex.
- λ and $\bar{\omega}$ are also dominated by an unquantifiable property of the structure of regex. To put it simple, regexes with longer but less closures are more likely to avoid the disadvantages and have higher accuracy and improve rate in repair than those who do not.
- The efficiency is affected by data size, average length of strings and error types. The run time is linear with M and L_s, while T has little impact on that. Additionally, RSR algorithm can repair 2,000 items within 5 seconds. That is, given a string and a regular expression, our approach only takes at most tens of ms, which shows that our approach is efficient and scalable. On the other hand, the optimization strategy shows acceptable performance on decreasing the run time, particularly in long data strings and regexes, and on maintaining effectiveness.

6. CONCLUSION

To repair the strings violating the regex, we propose the RSR algorithm in this paper. We also combine edit-distance-based approach and associate rules to determine the inserted or revised values. Experimental results demonstrate the efficiency and effectiveness of the proposed algorithm.

Our algorithm is suitable for large-scale regex-based data cleaning, which is an comparatively new topic with few efforts. The future work aims at better accuracy and efficiency. For accuracy, the optimization to overcome the disadvantages of RSR algorithm will be conducted. More flexible definitions about regexes can also help avoid such disadvantages. For efficiency, parallel computing can be introduced since the repairing process for individual strings or tokens are independent given the NFA of the pattern and string entries can be repaired via multiple processors.

7. ACKNOWLEDGEMENTS

This paper was supported by NGFR 973 grant 2012CB316200, NSFC grant 61472099, 61133002 and National Sci-Tech Support Plan 2015BAH10F01. Hongzhi Wang and Zeyu Li contributed to the work eqully and should be regarded as co-first authors. Hongzhi Wang is the corresponding author of this work.

8. REFERENCES

[1] Summary of movielens datasets. `http://files.grouplens.org/datasets/movielens/ml-1m-README.txt`.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[4] M. Arenas, L. E. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. P. Spinrad. Scalar aggregation in inconsistent databases. *Theor. Comput. Sci.*, 296(3):405–434, 2003.

[5] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *STOC*, pages 51–58, 2015.

[6] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755, 2007.

[7] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 1(1):1166–1177, 2008.

[8] D. Deng, G. Li, J. Feng, and W. Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936, 2013.

[9] X. L. Dong, L. Berti-Equille, and D. Srivastava. Integrating conflicting data: The role of source dependence. *PVLDB*, 2(1):550–561, 2009.

[10] X. L. Dong, L. Berti-Equille, and D. Srivastava. Truth discovery and copying detection in a dynamic world. *PVLDB*, 2(1):562–573, 2009.

[11] W. Eckerson. Data quality and the bottom line. *Journal of Radioanalytical & Nuclear Chemistry*, 160(4):355–362, 1992.

[12] W. Fan, F. Geerts, N. Tang, and W. Yu. Inferring data currency and consistency for conflict resolution. In *ICDE*, pages 470–481, 2013.

[13] A. Galland, S. Abiteboul, A. Marian, and P. Senellart. Corroborating information from disagreeing views. In *WSDM*, pages 131–140, 2010.

[14] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9):625–636, 2013.

[15] K. Lakshminarayan, S. A. Harp, R. P. Goldman, and T. Samad. Imputation of missing data using machine learning techniques. In *KDD*, pages 140–145, 1996.

[16] C. Mayfield, J. Neville, and S. Prabhakar. ERACER: a database approach for statistical inference and data cleaning. In *SIGMOD*, pages 75–86, 2010.

[17] I. O. Medicine, J. M. Corrigan, and M. S. Donaldson. To err is human: Building a safer health system. *Institute of Medicine the National Academies*, 7(4):245–246, 2000.

[18] V. Raman and J. M. Hellerstein. Potter's Wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[19] N. A. Setiawan, P. A. Venkatachalam, and A. F. M. Hani. Missing attribute value prediction based on artificial neural network and rough set theory. In *BMEI*, pages 306–310, 2008.

[20] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[21] T. Warren. Using regular expressions for data cleansing and standardization. http://www.kimballgroup.com/2009/01/using-regular-expressions-for-data-cleansing-and -standardization.