Maximal Format-Free Data Repair

Zijian Luo
University of Sydney
Sydney, Australia
lzij5923@uni.sydney.edu.au

Xi Wu
University of Sydney
Sydney, Australia
xi.wu@sydney.edu.au

Hong Jin Kang
University of Sydney
Sydney, Australia
hongjin.kang@sydney.edu.au

Alan Fekete
University of Sydney
Sydney, Australia
alan.fekete@sydney.edu.au

Rahul Gopinath
University of Sydney
Sydney, Australia
rahul.gopinath@sydney.edu.au

ABSTRACT

Modern data-processing pipelines rely on input records conforming to strict format specifications. In practice, however, data corruption can occur at numerous stages including data-entry error, corruption during input processing and retransmission, inconsistent formatting, and incompatible specifications. Such corrupted data can result in loss of records, reducing the accuracy of processing.

Rather than discarding corrupted records and losing valuable information, one can attempt to repair the data. Data-repair solutions such as *regular expression based repair* and *error-correcting parsers* require a specification to perform structural repairs.

Specification-free techniques such as *ddmax* and εRepair are limited in repair operations, repair location, and require specific parser properties that are often unavailable.

To tackle this challenge, we introduce βMax, a novel format-free data repair algorithm optimal with respect to the provided example data, with maximal data-recovery and minimal parser constraints.

Despite requiring less information than εRepair, βMax repairs 83% of all corrupt records—1.77× the rate achieved by εRepair—while using 27.7× fewer oracle calls.

KEYWORDS

Input repair, Error correction, Data cleansing

ACM Reference Format:
Zijian Luo, Xi Wu, Hong Jin Kang, Alan Fekete, and Rahul Gopinath. 2026. Maximal Format-Free Data Repair. In . ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

While processing data collated from diverse sources, one must often contend with malformed data. This corruption could be artifacts of inconsistent processing [21, 40, 47], hardware failures [23], truncated network transmission [10, 45], or even human errors in data entry [28, 46]. Even within a single organization, CSV file formats can be inconsistent, with dates and times frequently containing incompatible delimiters [25]. Over 99% of spreadsheets contain non-validated columns [18], with over 35% of the columns representing data as simply of *string* type, which can result in inconsistencies.

In a data processing pipeline, such parser inconsistencies can cause record processing to fail. Even if the inconsistency is in a single field in a tabular data structure, automatic recovery can be infeasible [42, 44], and manual repair can be costly and error-prone [36]. Hence, data records containing errors are often *cleansed* from the pipeline [22], resulting in severe data loss.

Hence, automatic data repair techniques are needed. Format based repair [1, 13, 16, 17, 26, 31] can be applied if a format is available. However, format specifications are often rare, especially for one-off application-specific formats that are typically used in the processing of tabular data such as CSV, Excel, TSV etc.

Researchers have explored specification-free alternatives to data repair. For example, *ddmax* [27] can fix malformed data records provided three constraints are satisfied: (1) a parser that can recognize valid (✓) and invalid (×) inputs, (2) a starting minimal subsequence of the input ($\emptyset$) that is accepted as a valid parse, and (3) the ability to add fragments (δs) to this input, resulting in progressively larger *parsable inputs* (Ξ_i) called *waypoints*. Given a data format that conforms to these requirements and a corresponding parser, starting from Ξ_0 which is $\emptyset^{\checkmark}$, *ddmax* will iteratively identify chunks (δ_i) from the given (malformed) input that can be added to Ξ while retaining parsability resulting in a Ξ_{max} representing maximal data recovery from the original input. That is, data-repair with *ddmax* requires a conforming data format that has an empty valid parse ($\emptyset^{\checkmark}$), parseable waypoints (${\Xi_i}^{\checkmark}$), and a corresponding parser. However, data formats such as addresses, dates, times, zip codes, and telephone numbers common in tabular data do not have a *minimal* valid input that correspond to $\emptyset$. Furthermore, *ddmax* is limited to simply *deleting* any input fragment it deems as part of the corruption, exacerbating data loss when corruption is misidentified.

εRepair is another algorithm for data repair with a different tradeoff. Unlike *ddmax*, εRepair does not require a starting empty sequence $\emptyset$, nor parsable waypoints Ξ_i. Instead, it assumes that the *location of the corruption* can be identified from parser feedback. This allows εRepair to repair inputs even when valid minimal sequence ($\emptyset$) or valid waypoints (Ξ_i) are unavailable. Furthermore, εRepair utilizes all corrections—*deletion*, *insertion*, and *replacement*—achieving better repair.

However, εRepair has two limitations: (1) It requires precise error location, which is often unavailable. (2) It conflates the location of minimal repair with the error location returned by the parser, which may not lead to minimal repairs.

To address these issues, we propose βMax, a novel approach for automatic data repair. The key innovation in βMax is to recognize that in most data-repair settings, there will be numerous inputs that are accepted by the parser program, and a few that are corrupted

Conference'17, July 2017, Washington, DC, USA
2026. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

and hence, rejected. This provides us with a set of valid and invalid examples to form an initial approximate specification [38], which is then iteratively refined. This approximate specification is used in conjunction with the error correcting parser [2] to produce an initial *repair* of a given corrupt input. The repair, if not accepted by the parser, is then used to refine the approximate specification, producing closer repairs iteratively until a repair is accepted.

βMax has two key advantages over εRepair: (1) Being independent of parser characteristics, it can be applied to any data format that has a simple validator. (2) Similar to numerical data-imputation, the repair provided is structurally closest to available samples, reducing impact of a potentially incorrect repair on the data set. βMax focuses on repairing individual fields in data records—dates, times, network addresses—which are typically regular languages validated by matchers offering only accept/reject feedback.

Although βMax operates with less information than εRepair, βMax performs 1.77× better than εRepair, repairing 83% of all corrupt records, while using 27.7× fewer oracle calls.

We make the following contributions:

- A novel specification-free data-repair algorithm expecting only a parser that can accept (✓) or reject (↯) a given input.
- Improvement in data-repair quality and performance compared to the state-of-the-art for regular formats.
- An empirical comparison of εRepair and βMax.

2 CHALLENGES IN DATA REPAIR

The problem of data repair is the following: Given a corrupt string, find the best repair such that the maximal amount of original data is saved [34]. For example, consider the JSON fragment below:

```
{"name": "John Doe", "email": "john@doe.com john@doe.org",
"phone": "2 8627 1444", "date": "2025/03/01",
"web": "http://f0f::]:80 http://[f0fhost.com:80" }
```

After the first level of parsing by a JSON parser, the individual fields will undergo further processing, for example, to validate *email*, *phone*, and *date*, each of which are specified as strings. Typically, this validation will be done using regular expressions or small custom functions. In this example, there is a problem: The *email* field contains two emails rather than the expected single email. Perhaps the emails should have been separated with '`;`' rather than a space. This is a relatively common error that can cause the parser to fail on this field, and reject the entire record. To recover the needed data, we need to repair this field. A similar issue applies to *phone* where spaces are not allowed. Similarly, the date format may be parsed by ISO8601 [24], which requires '`-`' as the delimiter.

We consider each state-of-the-art data repair methods in turn.

2.1 Data repair with *ddmax*

The *ddmax* algorithm is based on *ddmin*, and works as follows: ✓ denotes parser acceptance, × denotes rejection.

(1) *ddmax* begins with an empty valid input (∅). Let us call this input Ξ_0. The input can also be a minimal starting value. Such minimal values should conform to two requirements: (1) It should be acceptable as the smallest *valid* input for the given format. For example, Ξ_0 ={}✓ is a reasonable starting input for JSON objects. (2) It should be producible by deleting fragments from the original input. If not, the input cannot be repaired by *ddmax*.
(2) The *ddmax* algorithm then identifies fragments (named δ_i) in the original input that can be added to Ξ_i such that $\Xi_{i+1} = \Xi_i + \delta_i$✓. Any δ_i such that $\Xi_i + \delta_i$× is rejected. This process continues until all δ_i in the input is exhausted.
(3) This produces Ξ_{max}✓ representing the maximal repair.

Considering the malformed email field in the corrupt record, we find three impediments to applying *ddmax*. (1) There is no simple equivalent for an empty field (∅✓) for *phone* and *date*, each of which are fixed width fields. Hence, *ddmax* cannot be applied on such fields. (2) Even in variable width fields where one can find an empty field such as *email* which satisfies the parser (for example, ∅ = @) it can be non-obvious how to obtain that input, and in some cases, there may be multiple such inputs. For example, it is non-obvious what the minimal starting input should be for repairing `{"abc": []`. Doing this for thousands of diverse fields can be tedious and error-prone. (3) *ddmax* uses *deletion* as the only repair action. Deletion is insufficient to repair fixed width fields with data loss.

Hence, *ddmax* is unsuitable for data repair under such conditions. Let us now examine how εRepair fares on such malformed inputs.

2.2 Data repair with εRepair

εRepair works as follows: Let ✓ indicate an input accepted by the parser as before. However, the parser is also expected to distinguish when the *input* is a valid *prefix* of some valid string. For example, `[{"key":` is a valid JSON prefix, and is indicated by ▷. If the given text is not a valid prefix, for example, `[{"key":]` which does not have a valid completion, it is indicated by ↯. The ▷ signal can also be inferred if the parser reports the syntax error location.

Given a malformed input and a parser that can distinguish ▷ from ↯, εRepair proceeds as follows:

(1) **Boundary search.** εRepair begins by locating the position of the syntax error (called the *parse boundary*) using binary search. This boundary is used to initialize the thread queue with a single empty repair thread.
(2) **Apply Repair.** From any active repair thread, εRepair performs either a deletion or an insertion at the parse boundary, producing new repair threads:
  (a) A repair thread where a single character or token is deleted from the input at the error location, allowing extending the parse boundary using the original data.
  (b) Repair threads where a character is inserted at the error location for each character in the alphabet that yields an ▷ response, enabling the parse boundary to be extended using the original data.
(3) **Extend Boundary.** For each repair thread, the parse boundary is extended by appending remaining characters. Non-extendable threads are discarded.
(4) **Select Threads.** The algorithm selects the highest-priority repair thread from the priority queue (ordered by preserved data) and repeats from step 2. Duplicate threads are eliminated, retaining only those with the fewest repairs.
(5) **Check Completion.** When the parser signals (i.e. ✓) that a repair thread is complete, the process terminates and the repaired input is returned.

While εRepair handles structurally rich formats, its generality is limited when corruption occurs in fields validated by ad hoc matchers or when parser feedback is unreliable.

Limited usefulness of parser feedback. Corruptions in fields such as *email*, *phone*, or *date* are typically checked using regular expressions, and such validators offer only a match/non-match result. Without information about whether the input is merely incomplete or fundamentally incorrect, εRepair cannot infer a boundary to guide local edits. As a consequence, these common scalar fields fall entirely outside its repairable scope.

Sensitivity to error-boundary placement. εRepair assumes that the parser's error location reliably indicates where repairs should occur. This assumption can be misleading. For example, in `http://f00f:•:]:80`, the local boundary reported after `f00f:` directs εRepair toward deleting `:]:`, whereas the optimal fix is to insert `[` before `f00f`. Similarly, in `http://[f0f•host.com:80`, deleting the stray `[` is the correct solution. Yet it lies outside the parser-indicated region. In such cases, εRepair 's locality bias prevents it from recovering globally minimal repairs.

These examples illustrate a broader pattern: methods that depend on parse-boundary signals remain vulnerable to coarse, incomplete, or unhelpful parser diagnostics. This dependency makes it difficult to extend εRepair to formats validated only by regular expressions, or to parsers whose feedback cannot reliably guide structural edits.

These limitations point toward a natural design direction: if feedback cannot be trusted, then a repair technique must avoid relying on parser characteristics in the first place.

The next section introduces a complementary approach, called βMax, that removes this dependency. Unlike *ddmax* and εRepair, βMax grounds its repairs in provided example data rather than parser feedback, producing repairs that are optimal with respect to an iteratively refined grammar.

3 RECORD REPAIR WITH BMAX

We first define a few terms used in this section.

oracle A blackbox that accepts *string* inputs and returns × or ✓.
alphabet Symbols in the input language of the *oracle*.
string An ordered sequence of input symbols. A string is *valid*, or *accepted* if the *oracle* returns ✓ when given this string, and it is *invalid*, or *rejected* if the oracle returns ×.
grammar A grammar defines the input specification for an oracle.

Similar to specification-based data-repair [1, 13, 39], βMax assumes that the corrupted data originally conformed to some specification. Following numeric data imputation principles, adapted for non-numeric, structured data, βMax assumes that the original data is similar to the provided valid examples, and the repair algorithm's task is to produce a repair that is closest to these examples.

Consequently, βMax requires valid example data and, similar to both *ddmax* and εRepair, assumes that there exists an oracle that will respond ✓ to valid inputs, and × to invalid inputs. Note that βMax is designed for, and leverages *regular-language* inference.

Let us start with an oracle that accepts the *hostname* (RFC952 [20]). The βMax algorithm starts with a few examples: (1) `a`✓ (2) `ac`✓ (3) `b1e`✓ (4) `b1f`✓. An invalid string (5) `1f`× needs repair.

The top-level βMax algorithm is given in Algorithm 4. We describe each step in the subsections that follow.

3.1 Prefix-Tree Acceptor

βMax begins by creating a prefix-tree acceptor automata (PTA) that corresponds to the given valid strings (Figure 1). Creating such a PTA is the first step in grammar inference from labeled data [33].

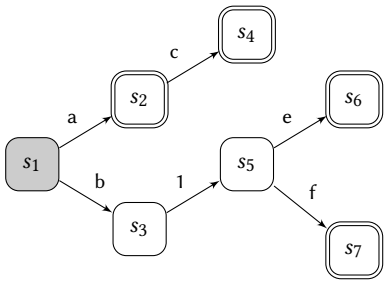

Figure 1: Prefix tree automata

$\langle s_1 \rangle \rightarrow$ a $\langle s_2 \rangle$ | b $\langle s_3 \rangle$
$\langle s_2 \rangle \rightarrow$ c $\langle s_4 \rangle$ | ϵ
$\langle s_3 \rangle \rightarrow$ 1 $\langle s_5 \rangle$
$\langle s_4 \rangle \rightarrow \epsilon$
$\langle s_5 \rangle \rightarrow$ f $\langle s_7 \rangle$ | e $\langle s_6 \rangle$
$\langle s_6 \rangle \rightarrow \epsilon$
$\langle s_7 \rangle \rightarrow \epsilon$

Figure 2: The automata as grammar

Each prefix maps to a unique state, and the PTA accepts the input strings exactly, capturing the most specific behavior consistent with these strings. The states in the PTA are ordered and numbered.

An automaton can be represented as a *regular grammar*. The starting state of the automaton becomes the start symbol of the regular grammar. Each state is represented by a nonterminal symbol in the grammar, and each transition is represented by a terminal symbol. The transitions from a state are represented by production rules in the equivalent grammar, starting with the transition symbol, and ending with the nonterminal symbol corresponding to the next state. The accepting state is represented by a nonterminal symbol that has empty string (ϵ) as the production rule. The regular grammar for the PTA in Figure 1 is given in Figure 2.

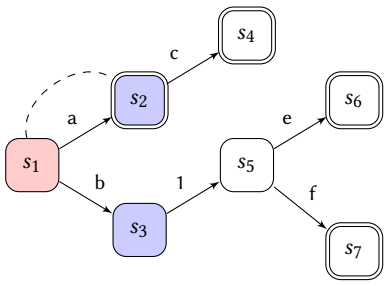

Figure 3: After step (1), (2), and (3)

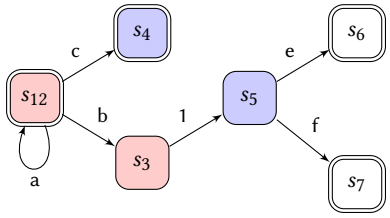

Figure 4: After step (4), (5), (2)

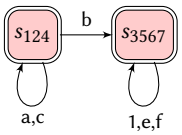

Figure 5: Automata A_1 without blue states

$\langle s_{124} \rangle \rightarrow \epsilon$
| a $\langle s_{124} \rangle$
| c $\langle s_{124} \rangle$
| b $\langle s_{3567} \rangle$
$\langle s_{3567} \rangle \rightarrow \epsilon$
| 1 $\langle s_{3567} \rangle$
| e $\langle s_{3567} \rangle$
| f $\langle s_{3567} \rangle$

Figure 6: G_1 grammar

$\langle s_{124} \rangle \rightarrow \epsilon$
| $\langle T_a \rangle$ $\langle s_{124} \rangle$
| $\langle T_c \rangle$ $\langle s_{124} \rangle$
| $\langle T_b \rangle$ $\langle s_{3567} \rangle$
$\langle s_{3567} \rangle \rightarrow \epsilon$
| $\langle T_1 \rangle$ $\langle s_{3567} \rangle$
| $\langle T_e \rangle$ $\langle s_{3567} \rangle$
| $\langle T_f \rangle$ $\langle s_{3567} \rangle$

Figure 7: G_1^{cover} grammar

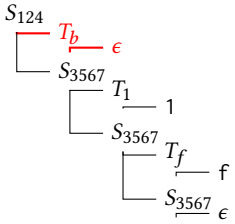

Figure 8: 1f× Parse tree.

Let us call this grammar G_0. From now on, we will assume that the automaton is represented as a regular grammar.

Augmenting invalid examples with neighborhood exploration. In traditional algorithms for inferring languages from labeled data [38], the labeled valid strings determine the effectiveness of *recall* of the final grammar. On the other hand, the effectiveness of *precision* is determined by the labeled invalid strings. We found that a single invalid string (the string to be repaired) was insufficient to generate grammars with sufficient precision and we needed additional invalid strings. While it is easy to generate invalid strings, we found that the *quality* of such strings were important. An invalid string is of high quality if it is very close to validity. That is, at most one or two symbols away from a valid string. Such invalid strings serve to constrain the generalization process, and hence improve the *precision* of the automaton produced.

To produce such strings, we mutated the PTA one transition at a time. Then, we produced strings that contained that transition. That is, if there are n transitions in the PTA, this process produces n new PTAs, from which we produce a new set of strings that differ from the original by exactly one character. Next, we check the strings generated by these new PTAs. If the string from the newly induced transition was rejected, it is added to the invalid examples. We call this process the *neighborhood exploration*, or *NE* for short.

In our case, our example set is extended by (6) 1e× and (7) 1g×. Note that the invalid samples do not change the PTA.

Algorithm 1 State Merging Algorithm

```
1:  function StateMerge(S+, S−, P)
2:      A ← PTA(S+)
3:      H ← []                          ▷ Record committed merges (in order)
4:      Red ← {q0}; Blue ← children(q0)
5:      while Blue ≠ ∅ do
6:          q_b ← select(Blue)
7:          S_b+ ← {x ∈ S+ | x passes through q_b}
8:          merged ← false
9:          for all q_r ∈ Red do
10:             S_r+ ← {x ∈ S+ | x passes through q_r}
11:             if canMerge(S_r+, S_b+, P) then
12:                 A' ← merge(A, q_r, q_b)
13:                 if verified(A', S+, S−, P) then
14:                     A ← A'
15:                     append(H, (q_r, q_b))
16:                     merged ← true
17:                     break
18:         if not merged then
19:             Red ← Red ∪ {q_b}
20:         Blue ← {c | s ∈ Red, c ∈ children(s), c ∉ Red}
21:     return A, H
```

Algorithm 2 Can Merge Check for Candidates

```
1:  function CanMerge(S_i+, S_j+, P)
2:      for sample((x_i, x_j) ∈ S_i+ × S_j+, K) do      ▷ We use K=1
3:          for (a, b) ∈ {(x_i, x_j), (x_j, x_i)} do
4:              if concat(prefix(a), suffix(b)) ∉ L(P) then
5:                  return false
6:      return true
```

Algorithm 3 Consistency Check for Candidate Merge

```
1:  function Verified(A, S+, S−, P)
2:      return ¬ any(w ∈ S− : A(w))
```

3.2 State Merging and Generalization

The next step is to detect if any pairs of states can be merged. Two states can be merged if and only if all inputs that were originally identified as invalid remain invalid. Any input that was previously valid continues to be valid when any two states in the automaton are merged. For this, we adapt the state merging algorithm from the classic RPNI blue-fringe variation [30, 38] as below.

(1) Mark the initial state as red (Figure 3).
(2) Identify all states that are reachable from the initial state in a single transition. Mark them as blue (Figure 3).
(3) Pick a blue state in a deterministic fashion (Figure 3).
(4) Try merging the selected blue state with each red state, one at a time. Merging two states produces a new state whose incoming and outgoing transitions are the unions of the respective transitions from both original states. Commit to the merge if the chosen merge is *consistent*, and continue with the next blue state in step 3 (Figure 4).
(5) If none of the red states could be merged with the picked blue state, then color it red. Continue to (2) if any blue states exist. Otherwise, output the last automaton (Figure 5).

Algorithm 4 βMax Repair Algorithm

```
function BetaMax(S+, c0, P)
    S− ← {c0} ∪ NE(S+, P)
    G, H ← StateMerge(S+, S−, P)
    loop
        cr ← MinPenaltyParse(Cover(G), c0)
        if P(cr) = ✓ then return cr
        S− ← S− ∪ {cr}
        G ← ReplayMerges(S+, S−, H, P)
```

Incremental refinement by merge replay. The initial call to StateMerge also records a *merge history H* consisting of all *committed* (successful) merges in their original order. When a candidate repair c_r is rejected, we add c_r as a new negative counterexample and *refine* the hypothesis by replaying only the merges in H. Any merge that would now accept a negative counterexample is skipped.

Checking mergeability. The quality of repairs is strongly influenced by the accuracy of the grammar inferred. βMax starts with a set of examples. The valid examples determine the *recall* of the inferred grammar, and the invalid examples determine the *precision*. To ensure that we do not merge states that should not be merged, we identify new strings that can be created by the merger of two states, and check their validity with the blackbox program.

Given a state A, any string X that passes through A can be split into X_{prefix} which is the path from the start state to A, and X_{suffix} which is the path from A to the final state. Two states A, and B, with strings X and Y passing through them, can be merged if and only if all possible strings (X_{prefix}, Y_{suffix}) are accepted by oracle. This algorithm is given in Algorithm 2. We found a sampling strategy of $K = 1$ optimal for data recovery and repair rate (Table 11). The state-merging algorithm is given in Algorithm 1.

Checking consistency. When merging states all invalid strings from the example set should also remain invalid after merge. For example, in the PTA from Figure 1, merging the states `s1` and `s2` results in a new automata (Figure 4) which continues to reject (5) `1f`$^{\times}$, (6) `1e`$^{\times}$, and (7) `1g`$^{\times}$. Similarly, the final automata in Figure 5 also continues to reject these inputs. This check is given in Algorithm 3.

This process results in a refined hypothesis automaton we call A_1. The corresponding regular grammar G_1 is given in Figure 6.

3.3 Covering Grammar and Repair

The G_1 grammar from Section 3.2 is next used to generate a potential repair for our corrupted data record. This covering-grammar approach used by structural repair [1, 41] is described below.

Given a grammar G_1, each terminal symbol x is replaced by a nonterminal symbol $\langle T_x \rangle$ with the following definition that provides tolerance for errors. Let 'Σ' denote any character, 'Σ^+' one or more characters, and let '$\Sigma_{\hat{x}}$' denote any character other than x. Then,

$$\langle T_x \rangle \rightarrow x \mid \Sigma^+ x \mid \Sigma_{\hat{x}} \mid \epsilon$$

Intuitively, each character x may be realized as (1) itself, (2) any non-empty sequence of characters ending in x (insertion), (3) a single character different from x (substitution), or (4) deletion. This gives us the covering grammar G^{cover}. Notably, the $\Sigma^+ x$ production permits multi-character insertions, handling two- or three-character corruptions in one step.

Next, production rules are annotated with penalties. Each use of an error-tolerant production rule such as $\Sigma^+ x$, $\Sigma_{\hat{x}}$, or ϵ incurs a positive penalty, whereas an exact match on x incurs zero penalty. We then use any context-free parser and parse the corrupted string c_0 with this grammar and choose candidate with the least penalty.

For example, parsing the corrupt input `1f`$^{\times}$ with G^{cover} produces the parse tree in Figure 8, which indicates the least number of edits to be made to the string to make it parsable by G_1. In the parse tree (Figure 8), we replace $T_b \rightarrow \epsilon$ with `b`, which is the only edit needed. We apply these edits, producing the repair candidate c_1.

This repair candidate is processed by the parser. In our example, the parse accepts `b1f`$^{\checkmark}$. However, if the parser rejects the repair candidate c_1, we add c_1 as a new counterexample and refine the hypothesis, producing the next approximation grammar G_2.

This process continues until the repair is complete.

The repair produced is optimal with respect to the inferred grammar: no correction consistent with the grammar incurs fewer edits. As the grammar is iteratively refined to approximate the target format, successive repairs converge closer to the example data.

4 EVALUATION

How does βMax compare with state-of-the-art? We propose the following research questions to investigate βMax effectiveness.

4.1 Research Questions

Among parser-centric, format-agnostic repair techniques, εRepair is state-of-the-art and uniquely handles records without identifiable waypoints, hence we use this as benchmark. Specification-based approaches such as RSR [31] are excluded as they are non-comparable.

Given the limited information available to βMax in comparison to εRepair, the first question we ask is about whether data repair can actually repair a significant number of inputs.

RQ1: How many inputs can βMax repair in comparison to εRepair? An answer to this question can tell us whether grammar inference can form a useful tool for a data engineer.

The next question we ask is about the quality of data repair achieved in *conserving* original data.

RQ2: What is the quality of data repair by βMax in comparison to εRepair?

Finally, a data repair algorithm should be feasible to run for thousands of corrupt records. Hence, we ask:

RQ3: How does βMax compare to other techniques in runtime performance and oracle calls required?

The details of experiments are given next.

Platform. Experiments were conducted on a Mac M2 Ultra with 192 GB RAM.

Research Protocol. We ran each repair technique and collected metrics, with a time limit of 300 seconds per record for both εRepair and βMax. Earley algorithm [6] wase used for parsing.

4.2 Evaluation Subjects

To enable accurate assessment, we evaluated βMax on six representative string-based input categories with regular expressions from regexlib.com: (1) `date` validating calendar dates, (2) `time` validating time stamps, (3) `isbn` validating ISBN identifiers, (4) `ipv4` for IPv4 addresses, (5) `ipv6` for IPv6 addresses, and (6) `url` for URLs.

Table 1: Subject regular expressions and average record lengths ($\mu|r|$) from 100 records.

Cat.	Regex	$\mu\|r\|$
Date	`^\d{4}-\d{2}-\d{2}$`	10.00 ± 0.00
Time	`^\d{2}:\d{2}:\d{2}$`	8.00 ± 0.00
ISBN	`^(?:\d[- ]?){9}[\dX]$`	13.00 ± 0.00
IPv4	`^(\d{1,3}\.){3}\d{1,3}$`	13.60 ± 1.16
IPv6	`^([0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$`	26.36 ± 3.00
URL	`^https?:\/\/(www\.)?[-a-zA-Z0-9@:%._\+~#=]{1,256}\.[a-zA-Z0-9()]{1,6}\b([-a-zA-Z0-9()@:%_+.~#?&//=]*)$`	32.81 ± 12.89

Table 2: Successfully repaired records across corruption levels (timeout = 300s, total records = 50 per subject per row).

	Subject	βMax	βMax %	εRepair	εRepair %
1-corruption	date	**50**	**100%**	15	30%
	ipv4	**50**	**100%**	40	80%
	ipv6	**36**	**72%**	35	70%
	isbn	**49**	**98%**	43	86%
	time	**47**	**94%**	14	28%
	url	28	56	**37**	**74%**
2-corruption	date	**50**	**100%**	13	26%
	ipv4	**36**	**72%**	37	74%
	ipv6	**21**	**42%**	32	64%
	isbn	**48**	**96%**	33	66%
	time	**50**	**100%**	9	18%
	url	**38**	**76%**	30	60%
3-corruption	date	**48**	**96%**	6	12%
	ipv4	**50**	**100%**	33	66%
	ipv6	**23**	**46%**	23	46%
	isbn	**48**	**96%**	17	34%
	time	**48**	**96%**	3	6%
	url	**31**	**62%**	6	12%
Total		**751**	**83.4%**	426	47%

Table 1 reports the regular expressions used for each category. We manually collected valid records for each of these categories. For each category, we curated a dataset of 100 valid examples. The average record lengths ($\mu|r|$) are shown in Table 1. We split these into two subsets randomly: 50 records for grammar inference (training) and 50 records for record repair (testing).

Single/Double/Triple corruptions For each record, we induced corruptions uniformly at random from three operations: deletion, substitution, or insertion. We evaluated three corruption levels: 1-corruption (single character), 2-corruption (two consecutive characters), and 3-corruption (three consecutive characters). If a corruption did not result in a parse error, we resampled until a valid corrupt record was obtained. For each format, we produced 100 records for each of the 1-, 2-, and 3-corruption levels.

5 RESULTS

RQ1: How many corrupt records can be repaired by βMax? We compare the number of repaired files for βMax and εRepair (Table 2). The best values per row are bolded. We also provide the percentage of inputs repaired by each approach.

RQ2: What is the quality of data repair by βMax and εRepair? We collected all corrupt records that could be successfully repaired by each technique. For each repaired pair, we computed the *edit distance* between the corrupt record and the repaired record; results are summarized in Table 3. The standard deviation is given after ±. The best values are marked in bold, with overall average last.

For a data engineer, the most important question in terms of data recovery is the amount of data that could be recovered from the given corrupted data. Table 4 captures the amount of data that was recovered from the corrupted data by each approach.

RQ3: How does βMax compare to εRepair in runtime performance and oracle calls required? We report average runtime in Table 5 and oracle calls in the repair phase in Table 6. βMax averages 35.8 seconds per record compared to 55.9 seconds for εRepair, a 1.6× speedup. More significantly, βMax invokes the oracle on average only 1.5 times per record compared to 41.5 times for εRepair—a 27.7× reduction. This advantage is especially important when oracle invocation is costly, for example when the validator is accessed over a network. We additionally report the one-time *precompute* oracle cost of constructing the initial hypothesis DFA (including neighborhood exploration via mutation augmentation and cross-merge validation during state merging). This hypothesis DFA (and its merge history) is cached and reused across different inputs. When a candidate repair is rejected, βMax refines the hypothesis DFA incrementally by replaying the recorded merge history under the expanded negative set, rather than re-running state merging from scratch. Therefore, the precompute oracle calls in Table 7 are amortized and incurred only once per subject. The detailed stage-wise breakdown of βMax execution time is given in Table 8 and discussed further in Section 6.

6 DISCUSSION

A significant number of real-world, especially tabular data formats are validated by regular expressions [15]. In many cases, it is impossible to modify the parser to provide an error feedback as εRepair requires. In such cases, one needs to rely on data-imputation from available example records. βMax shows that data-imputation from such available example records can be practical, and can even be faster than the state-of-the art baseline εRepair.

Our results from Table 3 show that βMax achieves an average edit distance of 2.8 compared to 1.7 achieved by εRepair. Indeed, having access to accurate error feedback allows εRepair to identify precisely where to repair so as to minimize the number of edits. However, even without access to parse feedback—we only know whether the data is accepted or not—βMax repairs come within one edit distance on average to that achieved by εRepair.

Critically, however, Table 4 shows that βMax recovers 92% of the original data compared to 91% from εRepair. This is important because the primary objective of a data engineer is to recover as much of the original data as possible.

Furthermore, βMax relies on existing data for the inferred grammar, which means that the repairs suggested by βMax are guided by existing data. In contrast, εRepair repairs are guided exclusively by parser feedback. Hence, βMax repairs will be closer to existing data records, and not unduly affected by parser idiosyncrasies.

Table 3: Levenshtein distances between corrupt and repaired data across corruption levels.

	Subject	βMax	εRepair
1-corruption	date	1.66 ± 1.42	**1.31 ± 0.46**
	ipv4	5.00 ± 3.36	**1.07 ± 0.26**
	ipv6	2.56 ± 2.03	**1.14 ± 0.35**
	isbn	**1.18 ± 0.39**	1.58 ± 0.49
	time	**1.53 ± 1.18**	1.43 ± 0.49
	url	1.75 ± 1.77	**1.05 ± 0.23**
2-corruption	date	**2.02 ± 0.68**	1.94 ± 0.56
	ipv4	1.69 ± 0.62	**1.65 ± 0.58**
	ipv6	2.90 ± 2.22	**1.81 ± 0.53**
	isbn	**1.90 ± 0.55**	2.09 ± 0.62
	time	1.92 ± 0.34	**1.86 ± 0.52**
	url	3.32 ± 3.06	**1.77 ± 0.42**
3-corruption	date	2.85 ± 0.54	**2.83 ± 0.37**
	ipv4	6.64 ± 3.45	**2.12 ± 0.69**
	ipv6	3.87 ± 2.29	**2.04 ± 0.69**
	isbn	**2.65 ± 0.63**	3.00 ± 0.69
	time	3.02 ± 0.69	**3.00 ± 0.00**
	url	4.10 ± 3.40	**2.50 ± 0.50**
Average		2.79 ± 2.35	**1.70 ± 0.71**

Table 4: Data recovery between corrupt and repaired data across corruption levels.

	Subject	βMax	εRepair
1-corruption	date	**91% ± 0%**	**91% ± 0%**
	ipv4	**91% ± 4%**	**91% ± 3%**
	ipv6	**97% ± 2%**	96% ± 0%
	isbn	**99% ± 2%**	92% ± 0%
	time	**89% ± 2%**	**89% ± 1%**
	url	94% ± 4%	**97% ± 1%**
2-corruption	date	**90% ± 5%**	89% ± 3%
	ipv4	88% ± 6%	**89% ± 5%**
	ipv6	**96% ± 2%**	94% ± 2%
	isbn	**98% ± 4%**	90% ± 4%
	time	**87% ± 7%**	86% ± 4%
	url	91% ± 5%	**97% ± 3%**
3-corruption	date	**87% ± 7%**	84% ± 5%
	ipv4	**87% ± 6%**	85% ± 5%
	ipv6	**96% ± 1%**	93% ± 2%
	isbn	**97% ± 4%**	84% ± 3%
	time	**81% ± 7%**	78% ± 2%
	url	91% ± 5%	**93% ± 2%**
Average		**92% ± 7%**	91% ± 5%

Table 5: Average total runtime.

	Subject	βMax	εRepair
1-corruption	date	**1.56±0.48**	7.31±10.48
	ipv4	50.24±59.71	**35.77±49.92**
	ipv6	148.99±78.62	**41.92±77.32**
	isbn	**16.66±38.98**	91.82±61.26
	time	**1.09±0.47**	59.75±83.45
	url	55.04±65.95	**52.10±76.04**
2-corruption	date	**3.51±0.37**	12.34±26.34
	ipv4	**35.43±72.20**	48.34±60.73
	ipv6	171.74±77.60	**45.24±79.13**
	isbn	**21.23±44.93**	94.20±58.51
	time	**1.41±0.10**	104.78±88.05
	url	**50.20±69.06**	135.91±91.28
3-corruption	date	**1.85±0.26**	120.43±75.93
	ipv4	90.49±55.76	**39.47±45.69**
	ipv6	128.41±80.97	**53.45±78.91**
	isbn	**14.72±33.95**	115.58±59.01
	time	**1.40±0.12**	161.03±93.51
	url	**38.22±54.43**	77.02±67.18
Overall avg.		**35.80±64.38**	55.94±70.19

Table 6: Oracle calls in the repair phase.

	Subject	βMax	εRepair
1-corruption	date	**1.00±0.00**	7.00±0.00
	ipv4	**1.00±0.00**	16.48±27.14
	ipv6	**1.94±1.00**	23.92±54.87
	isbn	**2.20±4.97**	61.23±46.38
	time	**1.00±0.00**	7.00±0.00
	url	**1.64±1.91**	15.47±30.68
2-corruption	date	**1.00±0.00**	68.38±45.87
	ipv4	**3.57±5.30**	33.94±42.83
	ipv6	**1.81±0.59**	33.23±60.02
	isbn	**2.33±3.78**	71.23±44.24
	time	**1.00±0.00**	79.00±64.94
	url	**1.00±0.00**	103.08±66.51
3-corruption	date	**1.00±0.00**	93.50±60.54
	ipv4	**1.72±0.45**	28.76±33.61
	ipv6	**1.65±0.63**	40.35±65.23
	isbn	**1.92±3.74**	82.29±42.86
	time	**1.00±0.00**	113.67±67.65
	url	**1.00±0.00**	53.50±47.21
Overall avg.		**1.50±2.32**	41.51±52.68

In terms of performance, Table 5 suggests that βMax has an average runtime of 35.8 seconds compared to 55.9 for εRepair. That is, βMax is 1.6× faster than εRepair in repairing data on average. Finally, we are leveraging an external program—the parser—for data-repair. Hence, it is instructive to look at how often this external program (the oracle) was invoked. Table 6 reports oracle calls in the repair phase. Our results show that, on average, βMax invoked the oracle only once. That is, after βMax inferred the base grammar from the provided examples, only a small correction was necessary on the grammar before the repaired input was accepted by the

Table 7: βMax precomputation oracle calls
Includes mutation augmentation and cross-merge validation.

Subject	#Oracle calls
date	1070
time	734
isbn	2751
ipv4	3903
url	854
ipv6	8566

Table 8: βMax stage-wise cost across corruption levels (iterations and time breakdown; timeout = 300s).

	Sub.	G. Inf (sec)	#Oracle calls	Orc. time (sec)	G. Repair (sec)	Total time (sec)
1-corruption	date	0.00 ± 0.00	1.00 ± 0.00	0.02 ± 0.00	0.88 ± 0.14	0.96 ± 0.14
	time	0.00 ± 0.00	1.00 ± 0.00	0.02 ± 0.00	0.36 ± 0.06	0.43 ± 0.06
	url	0.38 ± 1.15	1.64 ± 1.91	0.03 ± 0.04	79.73 ± 65.47	80.23 ± 65.84
	isbn	3.70 ± 15.60	2.20 ± 4.97	0.04 ± 0.09	10.27 ± 21.21	14.08 ± 36.87
	ipv4	2.83 ± 13.87	1.00 ± 0.00	0.02 ± 0.00	17.44 ± 1.57	20.36 ± 14.15
	ipv6	7.49 ± 8.62	1.94 ± 1.00	0.04 ± 0.02	141.25 ± 69.16	148.87 ± 73.89
2-corruption	date	0.00 ± 0.00	1.00 ± 0.00	0.02 ± 0.00	0.80 ± 0.22	1.89 ± 0.22
	time	0.00 ± 0.00	1.00 ± 0.00	0.02 ± 0.00	0.32 ± 0.10	1.41 ± 0.10
	url	0.00 ± 0.00	1.00 ± 0.00	0.02 ± 0.00	49.08 ± 69.04	50.20 ± 69.06
	isbn	7.53 ± 21.84	2.33 ± 3.78	0.04 ± 0.07	12.60 ± 23.07	21.23 ± 44.93
	ipv4	14.38 ± 31.85	3.57 ± 5.30	0.06 ± 0.10	19.93 ± 41.03	35.43 ± 72.20
	ipv6	15.82 ± 12.35	1.81 ± 0.59	0.03 ± 0.01	154.78 ± 68.25	171.74 ± 77.60
3-corruption	date	0.00 ± 0.00	1.00 ± 0.00	0.02 ± 0.00	0.78 ± 0.26	1.85 ± 0.26
	time	0.00 ± 0.00	1.00 ± 0.00	0.02 ± 0.00	0.34 ± 0.12	1.40 ± 0.12
	url	0.00 ± 0.00	1.00 ± 0.00	0.02 ± 0.00	37.10 ± 54.42	38.22 ± 54.43
	isbn	4.23 ± 15.97	1.92 ± 3.74	0.03 ± 0.07	9.41 ± 17.96	14.72 ± 33.95
	ipv4	74.53 ± 47.21	1.72 ± 0.45	0.03 ± 0.01	14.86 ± 8.76	90.49 ± 55.76
	ipv6	12.93 ± 13.20	1.65 ± 0.63	0.03 ± 0.01	114.37 ± 69.94	128.41 ± 80.97
Avg.		8.04 ± 24.92	1.50 ± 2.32	0.03 ± 0.04	27.01 ± 54.72	35.80 ± 64.38

G. Inf. = avg. grammar-inference time, Orc time = avg. time in oracle calls; G. Repair = avg. grammar-repair time, Total time = total time taken.

oracle. This is in contrast to εRepair which required 41.5 oracle calls on average. This is not surprising, however, because each thread in εRepair relies on oracle calls to provide guidance on what next to do. We observe here that the time taken by oracle calls is dependent on the subject and can vary widely. In particular, the validating parser may even be over the network, which can make oracle calls very expensive.

6.1 Ablation study of βMax performance.

Why does βMax require 35.8 seconds on average even though it only needs one oracle call? To investigate this, we decomposed the execution stages of βMax. This is provided in Table 8. The table shows that the biggest chunk of computation is spent on *grammar-repair* (G. Repair). That is, the regular grammar learned from *grammar inference* is extended into a covered grammar, and this grammar is used to parse the corrupted string. There are several potential ways to parse the string, all of which needs to be explored concurrently, and the best one with least penalty chosen, which induces the computational overhead.

Impact of initial positive examples. One of the questions that can be asked is about the impact of the initial seed corpus. In Table 9 we document the impact of initial seeds used (K). The

Table 9: Ablation study on the number of initial positive seeds (K), with corruptions combined (1, 2, and 3).

K	Edit distance	Avg. Repair rate
50	2.79 ± 2.35	**83.33%**
25	**2.47 ± 1.65**	72.11%
12	2.80 ± 1.87	79.11%
6	3.98 ± 3.97	82.78%

Edit distances are computed only over successfully repaired cases.

edit-distance column is the edit distance between corrupted and repaired strings. Our analysis reveals that $K = 50$ is the sweet-spot for the number of initial positive samples, which obtains the maximum number of repaired strings. When using valid examples more than this number, the initial automata produced becomes too complex, resulting in a much longer grammar-inference and grammar-repair time, reducing the success rate for larger samples. On the other hand, when using valid examples smaller than $K = 50$, the initial grammar inferred is rather simple, and multiple iterations are required to complete the repair. The edit distance is smaller at $K = 25$ because harder-to-repair corrupted strings were excluded from the successfully repaired set.

Impact of neighborhood exploration. Another question that we wanted to explore was about the impact of *neighborhood exploration*. How many new invalid strings should be constructed so that the repair quality is sufficient? To understand, we plotted the impact of different neighborhood (invalid) samples on repair quality. The result is given in Table 10.

Impact of active cross-merge validation A final question is on the impact of number of samples for cross-merge validation during state merging. The ablation study of 0 to 3 samples is provided in Table 11. Our results show that the maximum repair rate is achieved under 1 cross-merge validation sample, with a small (0.04%) decrease in recovery compared to no cross-merge validation at all.

Table 10: Ablation study: impact of data augmentation with NE on repair performance (mutations combined).

#NE	Edit distance	Avg iterations	Repair rate
0	2.09 ± 1.22	6.69 ± 23.32	67.44%
20	2.23 ± 1.29	**4.56 ± 17.57**	73.56%
40	2.71 ± 2.91	7.27 ± 23.56	70.78%
60	2.79 ± 2.35	8.04 ± 24.92	**83.33%**
80	**2.01 ± 1.17**	8.40 ± 24.24	73.00%
100	2.27 ± 1.54	5.59 ± 19.96	74.67%

The columns are as follows: *#NE* reports the number of invalid strings generated, *Edit-Distance* shows the average Levenshtein distance between corrupted and repaired strings, *Avg iterations* show the number of iterations necessary before the number of required invalid strings were generated, and *Repair rate* shows the number of successful repairs made.

Our analysis suggests that the repair rate peaks when the number of invalid strings is about 60. Hence, this is the number of invalid neighbors we used for empirical evaluation.

Table 11: Ablation study: impact of cross-merge validation (`Samples`) on repair performance (mutations combined).

Samples	Edit distance	Data recovery	Repair rate
0	3.45 ± 1.53	**90.48% ± 6.25%**	71.44%
1	2.49 ± 1.39	90.41% ± 6.73%	**83.78%**
2	**2.47 ± 1.37**	90.39% ± 6.80%	81.56%
3	2.54 ± 1.39	90.34% ± 6.70%	81.78%

6.2 Data-Imputation

How does our approach differ from regular-expression based structural repair methods such as RSR [31]? We note that each transition in the automaton we produce is based on at least one valid data record given. We only augment the set of invalid data records through mutation, and we intentionally do not attempt active grammar inference (e.g., with L*). This means that any data record produced will have values that are closest to other records in the data set. For example, it is likely that a host name field may accept `ipv4` or `ipv6` records. However, if all existing data is based on domain names, it is unlikely that a given record is an `ipv4` record even if the closest valid correction is to make the data an `ipv4` record.

6.3 Applicability beyond regular languages

How does βMax fare against εRepair when repairing complex data formats beyond regular expressions? We evaluated both on JSON, a context-free language, with results in Table 12 and Table 13. Even though βMax approximates the JSON format using a regular grammar, it achieves superior data recovery compared to εRepair across all corruption levels (95% vs 82% on average). The consistent data-recovery advantage suggests that regular-grammar approximation is a practical strategy even for context-free formats.

7 LIMITATIONS

Computational complexity of grammar-repair. The grammar-repair phase involves constructing and parsing with a covering grammar, which is inherently context-free. General context-free parsing has $O(n^3)$ time complexity with respect to input length. As shown in Table 8, the grammar-repair stage dominates execution time for longer inputs such as `IPv6` and `URL`. While this overhead is acceptable for typical data-record lengths encountered in tabular data, it can become unwieldy for substantially larger inputs. Optimizations such as more efficient parsing strategies may be necessary for scaling to longer records.

Dependence on initial sample quality. The performance of βMax strongly depends on the representativeness of the initial valid samples. As demonstrated in our ablation study (Table 9), both repair rate and recovery quality suffer when insufficient or non-representative samples are provided. If the initial corpus does not adequately capture the variability of the target format, the inferred grammar may be either too specific—failing to generalize to valid corrupted inputs—or too permissive after state merging—producing repairs that diverge from the original data. This is not a problem in practice where tabular data often contain a large number of valid records. If otherwise, practitioners should ensure that the seed corpus reflects the expected data record diversity.

Regular grammar assumption. βMax assumes that the data format accepted by the oracle can be approximated by a regular grammar. This assumption holds for many string-based formats commonly found in tabular data, including dates, times, identifiers, and network addresses. However, this assumption may not be valid for formats requiring context-free or context-sensitive grammars, such as nested JSON structures, XML documents, or programming language fragments. Similarly, binary formats such as TLV (Type-Length-Value) encodings, which encode structural constraints in field lengths, are not well-suited to our approach. Extending βMax to handle such formats would require fundamentally different grammar inference and repair mechanisms.

Oracle invocation constraints. βMax assumes that the oracle can be invoked programmatically and repeatedly at low cost. However, in certain practical settings, repeated oracle invocation may be infeasible. or prohibitively expensive. In such scenarios, a passive grammar inference approach with a more comprehensive initial corpus may be preferable.

8 RELATED WORK

We next document some of the closely related research.

Constraint-based Input Repair. Constraint-based input repair techniques restores invalid inputs by enforcing structural or semantic constraints. Such constraints may either be explicitly specified by developers or inferred automatically from a corpus of valid examples. Early work on data structure repair by Demsky and Rinard [11, 12] and subsequent extensions [43] focus on learning consistency invariants over data structures (e.g., pointer relationships or object graphs) and repairing violations to reestablish a consistent state. Related approaches in data cleaning and validation similarly rely on integrity constraints or rules learned from samples [32] or provided by users [14].

Specification-less black-box Input Repair. Specification-less black-box input repair techniques treat the target program or parser purely as an oracle that classifies inputs as valid or invalid without assuming access to internal program state, source code, or grammar specifications. *ddmax* [27] was the first black-box approach designed for input repair. It frames repair as a maximization problem: What is the maximal valid string that can be recovered from the corrupted input string? While effective in many scenarios, *ddmax* is inherently limited by its deletion-only repair model and its assumption that a minimal valid base input exists, and assumption of waypoints. Finally, *ddmax* can also result in extreme data-loss at times when corruptions interact with the *ddmax* partitioning [34].

εRepair [34] improves upon *ddmax* by incorporating insertion operations and by exploiting richer forms of parser feedback, enabling it to repair inputs that require the addition or modification of symbols rather than simple removal. However, such accurate feedback is not guaranteed in several places where input repair may be needed. While εRepair can repair data conforming to regular expressions, it requires specially instrumented regular expression library *RE2* for this purpose. This library may not be usable in all circumstances. To our knowledge, βMax is the only algorithm that can repair data that is described by regular expressions in blackbox settings without access to instrumented libraries.

Table 12: Levenshtein distances between corrupt and repaired data for JSON (timeout = 300s, total = 50 per row).

Corruption	βMax	εRepair
1-corruption	**2.12 ± 1.42**	2.68 ± 2.80
2-corruption	3.49 ± 1.13	**3.30 ± 2.30**
3-corruption	**3.94 ± 1.04**	5.10 ± 2.87
Average	3.18 ± 1.20	**3.69 ± 2.66**

Table 13: Data recovery between corrupt and repaired data for JSON (timeout = 300s, total = 50 per row).

Corruption	βMax	εRepair
1-corruption	**97% ± 6%**	88% ± 25%
2-corruption	**94% ± 7%**	85% ± 19%
3-corruption	**94% ± 7%**	72% ± 28%
Average	**95% ± 7%**	82% ± 24%

White and Gray-box Input Repair. White- and gray-box input repair techniques leverage varying degrees of access to the internal behavior of the program while processing the input. *Docovery* [29] uses symbolic execution to explore alternative execution paths and automatically modify corrupted documents so that the program avoids error states. Other techniques identify and analyze the specific regions of the input responsible for failures [3], enabling targeted modifications to those regions. Such approaches can produce precise repairs by understanding *why* an input fails, rather than merely observing *that* it fails. However, in contrast to βMax, these methods require access to program internals, source code, or sophisticated analysis infrastructure, which may not be available or practical for arbitrary parsers or large collections of programs.

Parser-directed Input Repair. Parser-directed input repair originates from compiler theory and focuses on recovering from syntax errors during parsing in order to continue processing the input. Classical techniques include minimum-distance error-correcting parsers [1], heuristic-based recovery strategies for LL and LR parsers [4, 5, 7, 9], as well as forward and backward recovery moves [8, 35]. Panic-mode recovery, which skips input symbols until a synchronization point is reached, is widely used in practice and surveyed in detail by Hammond and colleagues [19].

Modern parser generators such as ANTLR [39] incorporate sophisticated error recovery mechanisms, including token insertion, deletion, and replacement, to ensure that parsing can proceed even in the presence of errors. Recent work has revisited parser error recovery with a focus on improving diagnostic quality and minimizing unintended edits [13]. These approaches typically require access to an explicit grammar and prioritize continued parsing over faithful data recovery. As a result, the repaired input may be syntactically valid but semantically distant from the original data. In contrast, βMax aims to compute minimal-edit repairs guided by parser feedback, without necessarily requiring an explicit grammar, and with an emphasis on maximally preserving the original input.

LLM-based Input Repair. Recent work explores large language models (LLMs) for input repair and data cleaning [37, 48, 49]. LLM-based approaches are particularly effective for common and well-represented formats such as `INI`, `JSON`, and `SExp`, especially when similar data appears frequently in training corpora. By leveraging contextual understanding and learned patterns, LLMs can often propose plausible repairs for both syntactic and semantic errors.

Despite these advantages, our focus remains on algorithmic repair techniques: they offer deterministic, reproducible behavior, stronger correctness guarantees, and are free from hallucination or prompt sensitivity. They also integrate naturally into LLM pipelines as validation or minimization tools, supplying provably optimal solutions for downstream processing.

9 THREATS TO VALIDITY

External Validity. While we evaluated βMax across seven data formats, this covers only a narrow slice of the formats encountered in practice. βMax operates at the character level over an ASCII alphabet, so formats requiring Unicode or token-level representations fall outside its current scope. The corruption model used in our study may also not capture all real-world corruption patterns.

Internal Validity. Bugs in our implementation or inaccuracies in the baseline reimplementations could introduce bias. To reduce this risk we validated against known repair cases and relied on publicly available or carefully reconstructed baselines, though residual errors cannot be ruled out. All benchmarks were run on an Apple M2 Ultra workstation, so results on lower-powered hardware may differ. The 300-second per-repair time limit was fixed without a sensitivity study; tighter or looser budgets could shift both repair rate and edit-distance outcomes. Additionally, βMax stops at the first accepted repair rather than exhaustively searching for the best one, which may occasionally globally suboptimal result.

Construct Validity. Repair quality is quantified through edit distance, data-recovery rate, and parser acceptance. Although these metrics are well-established, they reflect structural fidelity rather than semantic correctness or suitability for downstream use. No human or domain-expert judgement was collected, which constrains how confidently the results can be interpreted in application contexts. Future work should address these gaps through user studies and broader downstream evaluations.

10 CONCLUSION

This paper presents βMax, a novel specification-free algorithm for repairing corrupted data records, producing repairs optimal with respect to the provided example data. By overcoming the limitations of both specification-dependent methods and existing specification-free techniques, βMax enables more flexible and effective structural recovery of malformed inputs.

Experimental results demonstrate clear improvements over the state-of-the-art εRepair, achieving 1.77× repair ability, while reducing oracle calls 27.7× compared to εRepair, and maintaining the quality of data recovery at 92%. These findings establish βMax as a robust and practical solution for data repair in real-world data-processing pipelines, particularly in environments where formal specifications or strict parser assumptions are unavailable but sufficient examples exist to guide the repair process.

DATA AVAILABILITY

All data for this research is available at:

https://doi.org/10.5281/zenodo.19122956

REFERENCES

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

[2] Alfred V Aho and Jeffrey D Ullman. 1971. The care and feeding of LR(k) grammars. In *ACM Symp. Theory Comput. (STOC)*. 159–170.

[3] Paul Eric Ammann and John C Knight. 1988. Data diversity: An approach to software fault tolerance. *IEEE Trans. Comput.* 4 (1988), 418–425.

[4] S. O. Anderson and Roland Carl Backhouse. 1981. Locally least-cost error recovery in Earley's algorithm. *ACM Trans. Program. Lang. Syst.* 3, 3 (1981), 318–347.

[5] S. O. Anderson, Roland Carl Backhouse, E. H. Bugge, and C. P. Stirling. 1983. An assessment of locally least-cost error recovery. *Comput. J.* 26, 1 (1983), 15–24.

[6] John Aycock and R Nigel Horspool. 2002. Practical Earley parsing. *Comput. J.* 45, 6 (2002), 620–630.

[7] Roland C Backhouse. 1979. *Syntax of Programming Languages: Theory and Practice*. Prentice-Hall.

[8] Michael Burke and Gerald A. Fisher. 1982. A practical method for syntactic error diagnosis and recovery. *ACM SIGPLAN Not.* 17, 6 (1982), 67–78.

[9] Carl Cerecke. 2003. *Locally least-cost error repair in LR parsers*. Ph. D. Dissertation. University of Canterbury.

[10] Data Core. 2021. Combatting Data Corruption in the Digital Age. https://www.datacore.com/glossary/data-corruption/.

[11] Brian Demsky and Martin Rinard. 2003. Automatic Detection and Repair of Errors in Data Structures. *ACM SIGPLAN Not.* 38, 11 (2003), 78–95.

[12] Brian Demsky and Martin Rinard. 2005. Data structure repair using goal-directed reasoning. In *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*. 176–185. https://doi.org/10.1109/ICSE.2005.1553560

[13] Lukas Diekmann and Laurence Tratt. 2020. Don't Panic! Better, Fewer, Syntax Errors for LR Parsers. In *Proc. Eur. Conf. Object-Oriented Program. (ECOOP) (LIPIcs, Vol. 166)*. Schloss Dagstuhl, 6:1–6:32.

[14] Bassem Elkarablieh and Sarfraz Khurshid. 2008. Juzi: A tool for repairing complex data structures. In *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*. ACM, 855–858.

[15] Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Rüemmer, and Andrei Sabelfeld. 2023. Black Ostrich: Web Application Scanning with String Solvers. In *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*. 549–563.

[16] Charles Fischer, Bernard Dion, and Jon Mauney. 1979. *A locally least-cost LR-error corrector*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[17] Charles N Fischer and Jon Mauney. 1992. A simple, fast, and effective LL(1) error repair algorithm. *Acta Informatica* 29, 2 (1992), 109–120.

[18] Marc Fisher and Gregg Rothermel. 2005. The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *Workshop on End-User Software Engineering*.

[19] K Hammond and Victor J. Rayward-Smith. 1984. A survey on syntactic error recovery and repair. *Computer Languages* 9, 1 (1984), 51–67.

[20] K. Harrenstien, M.K. Stahl, and E.J. Feinler. 1985. DoD Internet Host Table Specification. https://www.rfc-editor.org/rfc/rfc952.

[21] Joseph M Hellerstein. 2013. *Quantitative data cleaning for large databases*. https://dsf.berkeley.edu/jmh/papers/cleaning-unece.pdf

[22] Mauricio A Hernández and Salvatore J Stolfo. 1998. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery* 2 (1998), 9–37.

[23] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. 2021. Cores that don't count. In *Workshop on Hot Topics in Operating Systems (HotOS)*.

[24] ISO. [n. d.]. *ISO 8601*. https://en.wikipedia.org/wiki/ISO_8601

[25] Mujib Jimoh. 2023. *Date Formats: A Rare Headache in the Construction of Contractual Documents*. SSRN.

[26] Ik-Soon Kim and Kwangkeun Yi. 2010. LR error repair using the A* algorithm. *Acta Informatica* 47, 3 (2010), 179–207.

[27] Lukas Kirschner, Ezekiel Soremekun, and Andreas Zeller. 2020. Debugging Inputs. In *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*. 75–86.

[28] Marcin Kozak, Wojtek Krzanowski, Izabela Cichocka, and James Hartley. 2015. The effects of data input errors on subsequent statistical inference. *Journal of Applied Statistics* 42, 9 (2015), 2030–2037.

[29] Tomasz Kuchta, Cristian Cadar, Miguel Castro, and Manuel Costa. 2014. Docovery: Toward Generic Automatic Document Recovery. In *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*. 563–574.

[30] Kevin J Lang, Barak A Pearlmutter, and Rodney A Price. 1998. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In *Int. Colloq. Grammatical Inference (ICGI)*. Springer, 1–12.

[31] Zeyu Li, Hongzhi Wang, Wei Shao, Jianzhong Li, and Hong Gao. 2016. Repairing data through regular expressions. *Proc. VLDB Endow.* 9, 5 (2016), 432–443.

[32] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. 2012. Automatic Input Rectification. In *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*. IEEE Press, 80–90.

[33] Damián López and Pedro García. 2016. On the inference of finite state automata from positive and negative data. *Topics in Grammatical Inference* (2016), 73–112.

[34] Zijian Luo, Lukas Kirschner, Ezekiel Soremekun, and Rahul Gopinath. 2025. Automatic Data Repair without Format Specifications. In *Proc. IEEE Int. Symp. Softw. Reliab. Eng. (ISSRE)*.

[35] Jon Mauney and Charles N. Fischer. 1982. A forward move algorithm for LL and LR parsers. *ACM SIGPLAN Not.* 17, 6 (1982), 79–87.

[36] Mashaal Musleh, Mourad Ouzzani, Nan Tang, and AnHai Doan. 2020. CoClean: Collaborative Data Cleaning. In *Proc. ACM SIGMOD Int. Conf. Manag. Data*. 2757–2760.

[37] Wei Ni, Kaihang Zhang, Xiaoye Miao, Xiangyu Zhao, Yangyang Wu, and Jianwei Yin. 2024. IterClean: An Iterative Data Cleaning Framework with Large Language Models. In *ACM Turing Award Celebration Conference — China (ACM-TURC)*. 100–105.

[38] José Oncina and Pedro Garcia. 1992. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*. World Scientific, 99–108.

[39] Terence Parr and Kathleen Fisher. 2011. LL(*): The foundation of the ANTLR parser generator. *ACM SIGPLAN Not.* 46, 6 (2011), 425–436.

[40] Stephen G. Powell, Kenneth R. Baker, and Barry Lawson. 2008. A critical review of the literature on spreadsheet errors. *Decision Support Systems* (2008).

[41] S. Rajasekaran and M. Nicolae. 2016. An Error Correcting Parser for Context-Free Grammars that Takes Less Than Cubic Time. In *Proc. Int. Conf. Lang. Autom. Theory Appl. (LATA) (Lecture Notes in Computer Science, Vol. 9618)*. Springer, 533–546.

[42] Fakhitah Ridzuan and Wan Mohd Nazmee Wan Zainon. 2019. A review on data cleansing methods for big data. *Procedia Computer Science* 161 (2019), 731–738. https://doi.org/10.1016/j.procs.2019.11.177

[43] Martin C. Rinard. 2007. Living in the Comfort Zone. In *Proc. ACM Conf. Object-Oriented Program. Syst. Lang. Appl. (OOPSLA)*. ACM, 611–622.

[44] Christopher Scaffidi, Brad Myers, and Mary Shaw. 2008. Topes. In *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*. IEEE, 1–10.

[45] Christopher Scaffidi and Mary Shaw. 2008. Accommodating data heterogeneity in ULS systems. In *Workshop on Ultra-Large-Scale Software-Intensive Systems*. 15–18.

[46] Charles Cresson Wood and William W. Banks. 1993. Human error: an overlooked but significant information security problem. *Computers & Security* 12, 1 (1993).

[47] Mark Woodard, Sahra Sedigh Sarvestani, and Ali Hurson. 2015. A Survey of Research on Data Corruption in Cyber-Physical Critical Infrastructure Systems. *Advances in Computers* (2015).

[48] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. 2025. Aligning the Objective of LLM-based Program Repair. In *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*.

[49] Shuo Zhang, Zezhou Huang, and Eugene Wu. 2024. Data cleaning using large language models. *arXiv preprint arXiv:2410.15547* (2024).