Automatic Data Repair without Format Specifications

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

Lukas Kirschner
Saarland University
Germany
kirschlu@gmail.com

Ezekiel Soremekun
Singapore University of Technology and Design
Singapore
ezekiel_soremekun@sutd.edu.sg

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

***Abstract*—In data processing, datasets are expected to adhere to specific formats. However, inconsistencies due to human error, data corruption, or partial transmission can render these datasets nonconforming, hindering automated processing. This necessitates manual *data repair*, a time-consuming and error-prone task, especially when formal specifications are unavailable.**

To address this challenge, we introduce ϵREPAIR, a novel format-free approach to automating data repair. ϵREPAIR leverages parser feedback to detect and correct data inconsistencies, making it a versatile solution for data cleansing.

In evaluation, ϵREPAIR achieves 2.6× higher-quality repairs than its closest competitor, *DDMax*, in terms of the number of edits required to restore corrupted data, while reducing data loss by 2.8× compared to *DDMax*, with only 1.4×runtime overhead.

This work presents a practical, robust, and flexible format-free data repair alternative to *DDMax*. Its applications extend to domains such as data science, software development, and other human-centric systems, where handling diverse and inconsistent datasets is critical.

***Index Terms*—Input-repair, Error-correction, Data-Cleansing**

I. Introduction

Data repositories frequently contain corrupted records caused by unintended errors. These corruptions may occur during data creation due to human mistakes [1], [2], software bugs [3], [4], or hardware failures [5]. Corruptions can also arise during data modification—whether through interference by external actors or incomplete network transmission [6], [7].

For example, manual data entry without proper validation can introduce *nonconforming records* [8]. Spreadsheets—the most widely used reporting platform in the world [9]— contain over 40% textual data (excluding dates and times) [10], and more than 94% of them include errors [11]. Format inconsistencies across data sources can also result in incompatible implementations. For instance, different JSON libraries may interpret the same format in conflicting ways [12], [13], and database systems often support different SQL dialects [14]. Even within a single organization, CSV files can exhibit significant structural variation [15], while formats such as dates and times frequently use diverse standards and inconsistent delimiters across sources [16].

While formats like XML are considered standardized and rigid, a record may still be unparsable if the parser supports only a subset of the standard [17]. Faced with nonconforming records that are *almost*, but *not quite*, parsable, developers often carry the burden of *data repair*—a critical responsibility due to the prevalence of such records in data science and software engineering [18], [19]. Automatic data recovery is often infeasible [18], [20], leaving developers to either discard corrupted records [21] or manually repair them [18]—a time-consuming and error-prone task [22].

To address these issues, researchers have developed *automatic data repair* techniques, such as *error-correcting parsers* [23]–[25]. These approaches rely on formal specifications (i.e., grammars) to repair nonconforming records. However, their effectiveness is limited by the availability and reliability of such formal specifications. Many formats (e.g., CSV [15], WHATWG URL standards [26], [27], Markdown [28]) either lack formal specifications or have numerous conflicting versions. Moreover, parsers are often custom-built [29], [30], implementing subtle deviations from their documented specifications, which can make grammar-based methods suboptimal for real-world scenarios.

When formal specifications are unavailable, *DDMax* [19] remains the most effective known alternative. *DDMax* operates similarly to Delta Debugging [31], but in reverse. It requires three key components: (1) a parser to identify valid inputs, (2) a *starting minimal subsequence* of the input that is *valid*, and (3) the ability to add fragments (δs) to this input, resulting in larger *valid* inputs (called *waypoints*).

Given these prerequisites and a corrupt record that induces a parse error, *DDMax* proceeds as follows: (1) First, it identifies the fragments (δs) from the original record that can be added to the minimal valid subsequence without triggering a parse error. (2) Next, *DDMax* generates increasingly larger valid δ sequences by systematically adding more fragments and checking whether the resulting input remains valid according to the parser. (3) Finally, *DDMax* identifies the largest valid sequence of δs that can be inserted into the minimal subsequence without causing a parse error, and selects this sequence as the repaired record.

By identifying and maximizing the sections of the record that contain valid data fragments, *DDMax* isolates the portion of the record responsible for triggering the parse error, which is then eliminated. This process enables *DDMax* to maximize data retention while minimizing data loss.

However, *DDMax* has notable limitations: (1) *DDMax* only allows for fragment deletion (δs) as a repair operation, which is often suboptimal in practice. For example, in the case of

TABLE I: *DDMax* vs. ϵREPAIR: examples showing limitations of *DDMax* and the strengths of ϵREPAIR

Examples of Corrupt Inputs	***DDMax* Result**	**ϵREPAIR Result**	***DDMax* Limitation**
`{ "name": "Dave" "age": 42 }`	`42`	`{ "name": "Dave" ,"age": 42 }`	Limited repair options (deletion)
`{ "item": "Apple", "price" : ***3.45}`	`3.45`	`{ "item": "Apple", "price" : 3.45}`	Rich structure (spans)
`{"ABCD":[*"1,2,3,4,5,6"]*}`	`123456`	`{"ABCD":["1,2,3,4,5,6"]}`	Rich structure (multiple-faults)
`2024/07/23T12:34:56Z`	None	`2024-07-23T12:34:56Z`	Requires valid empty data

interrupted network transfers, data is typically truncated. In such cases, deleting additional data only worsens the problem. (Indeed, deleting parts of the surviving data in a network transfer is especially detrimental because the correctness of the transferred data is typically guaranteed by TCP/IP.) Similarly, in the case of disk corruption, the issue is not missing data but rather corrupt data. Here, the optimal repair would involve correcting the corrupted fragments where possible, rather than simply deleting them. Because *DDMax* only supports deletion, it cannot perform such replacements, limiting its effectiveness. (2) *DDMax* requires a *valid empty record* to start from, as well as valid records as waypoints to the maximally valid record. This is problematic in common data formats, such as date and time, which are typically defined using regular expressions and do not permit empty or incrementally valid forms. (3) In many cases, *DDMax* struggles with repairing multi-character corruptions because the partitioning strategy used by *DDMax* can't correctly isolate faulty fragments.

These constraints hinder *DDMax*'s ability to achieve maximal data repair, often resulting in considerable data loss. Table I illustrates a few such examples.

To overcome these shortcomings, we introduce ϵREPAIR, a novel approach leveraging the *parse-failure feedback* from parsers. This feedback often indicates the parse error location, precisely guiding data repair. Unlike *DDMax*, ϵREPAIR supports deletions, insertions, and replacements[1], enabling comprehensive and effective data repair.

ϵREPAIR shares foundational requirements such as the ability to validate records with *DDMax*, but removes the need for a valid *starting minimal subsequence* and intermediate valid *waypoints*. Instead, it relies on the parser to indicate location of the error, or at least whether a record is *incomplete* (a valid prefix of a conforming record) or *incorrect* (no suffix can make the record valid). This information is available from a wide variety of parsers [32]–[34] or can be externally incorporated (e.g., Mathis et al. [35]).

This paper makes the following contributions:

- **Identifying *DDMax* Limitations.** We highlight and demonstrate the shortcomings of *DDMax*, which can result in significant data loss.
- **Relaxed Requirements.** Our approach eliminates the need for *empty passing subsequence* and intermediate *valid waypoints* by leveraging parse-failure feedback.
- **Universal Data Repair.** In contrast to *DDMax*, our data repair methodology incorporates deletions, insertions, and consequently substitutions.
- **Empirical Evaluation.** We evaluate ϵREPAIR using 400 records belonging to four well-known data formats resulting in 4800 corrupted records. Our results show that in comparison to *DDMax*, ϵREPAIR delivers 2.6× higher-quality repairs, reducing data loss by 2.8×. We also demonstrate data repair on data described by regular expressions, a feat impossible for *DDMax*.

We structure the remainder of the paper as follows: Section II discusses *DDMax* limitations. Section III describes the ϵREPAIR algorithm. Our research questions are in Section IV. The results are in Section V, and we discuss the implications in Section VI. The related research is discussed in Section VII, threats in Section VIII, and Section IX concludes.

II. LIMITATIONS OF *DDMax*

We will first illustrate the limitations of the *DDMax* algorithm [19] and show how ϵREPAIR addresses these limitations.

A. Corrections in DDMax formalization

We identified two (minor) errors in the formal definition of *DDMax* (see Figure 1). Specifically, (1) *DDMax* requires the base case when $|c_\times - c_\checkmark| = 1$, where $c_\times$ is the failing configuration, and $c_\checkmark$ the passing configuration and the difference between the two is just a single character. If not, *DDMax* goes into unbounded recursion on inputs such as the incorrect ($c_\times$) JSON input: `1*1` where the closest $c_\checkmark$ is `11`. (2) When increasing granularity, the size of the remaining input should be considered instead of the entire text. Not doing this would cause an invariant failure for JSON inputs such as `{*"":2}`. The *DDMax* algorithm from Kirschner et al. [19, Figure 5] with corrections mentioned here in red is provided in Figure 1.

B. Limitations due to multiple faults

A *DDMax* failure pattern occurs for inputs with multiple errors. For example, consider the JSON input `[*]+`. Here, the JSON string is invalid because of two invalid characters that are non-contiguous. *DDMax* proceeds as follows:

1) The operation starts with $DDMax_2(\emptyset, 2)$
2) $|c_\times - \emptyset| \neq 1$. Hence, the base case does not apply
3) Can we increase to complement?
   $c_\times - \Delta_1$= `]+` ×
   $c_\times - \Delta_2$= `[*` ×

[1]Replacements are modeled as deletion followed by an insertion.

Maximizing Delta Debugging Algorithm

Let *test* and $c_\times$ be given such that $\mathit{test}(\emptyset) = \checkmark \wedge \mathit{test}(c_\times) = \times$ hold.
The goal is to find $c'_\checkmark = \mathit{DDMax}(c_\times)$ such that $c'_\checkmark \subset c_\times$, $\mathit{test}(c'_\checkmark) = \checkmark$, and $\Delta = c_\times - c'_\checkmark$ is 1-minimal.
The *maximizing Delta Debugging algorithm* $\mathit{DDMax}(c)$ is

$$\mathit{DDMax}(c_\times) = \mathit{DDMax}_2(\emptyset, 2) \quad \text{where}$$

$$\mathit{DDMax}_2(c'_\checkmark, n) = \begin{cases} c'_\checkmark & \text{if } |c_\times - c'_\checkmark| = 1 \text{ (“base case}^{a}\text{”)} \\ \mathit{DDMax}_2(c_\times - \Delta_i, 2) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot \mathit{test}(c_\times - \Delta_i) = \checkmark \\ & \text{(“increase to complement”)} \\ \mathit{DDMax}_2\big(c'_\checkmark \cup \Delta_i, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot \mathit{test}(c'_\checkmark \cup \Delta_i) = \checkmark \\ & \text{(“increase to subset”)} \\ \mathit{DDMax}_2\big(c'_\checkmark, \min(|c_\times - c'_\checkmark|, 2n)\big) & \text{else if } n < |c_\times - c'_\checkmark| \text{ (“increase granularity}^{b}\text{”)} \\ c'_\checkmark & \text{otherwise (“done”).} \end{cases}$$

where $\Delta = c_\times - c'_\checkmark = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c_\times - c'_\checkmark|/n$ holds.
The recursion invariant (and thus precondition) for DDMax_2 is $\mathit{test}(c'_\checkmark) = \checkmark \wedge n \leq |\Delta|$.
a: Bugfix: This base case is necessary to ensure that repairing JSON input `1*1` does not violate the invariant $n \leq |\Delta|$.
b: Bugfix: We should look for minimum of the remaining so that $n \leq |\Delta|$ is not violated for JSON input `{*"":2}`.

Fig. 1: Maximizing Lexical Delta Debugging algorithm from Kirschner et al. [19, Figure 5] with corrections in red.

4) Can we increase to subset?
   $\emptyset \cup \Delta_1$=`[*` ×
   $\emptyset \cup \Delta_2$= `]+` ×
5) Can we increase granularity? $n < |c_\times - c'_\checkmark|$
   which is $2 < |c_\times - \emptyset|$ ✓
   Hence the next iteration is: $\mathit{DDMax}_2\big(\emptyset, 4\big)$
6) $|c_\times - \emptyset| \neq 1$. Hence, the base case does not apply.
7) Can we increase to complement?
   $c_\times - \Delta_1$= `*]+` ×
   $c_\times - \Delta_2$= `[]+` ×
   $c_\times - \Delta_3$= `[*+` ×
   $c_\times - \Delta_4$= `[*]` ×
8) Can we increase to subset?
   $\emptyset \cup \Delta_1$=`[` ×
   $\emptyset \cup \Delta_2$=`*` ×
   $\emptyset \cup \Delta_3$=`]` ×
   $\emptyset \cup \Delta_4$=`+` ×
9) Can we increase granularity? $4 < 4$ ×
10) The solution is $\emptyset$.

That is, *DDMax* is unable to optimally repair inputs of this kind which contain multiple errors. To illustrate how this can lead to large data losses, consider Table I row 3. Here, `{"ABCD":[*"1,2,3,4,5,6"]*}` contains two distinct corruptions. *DDMax* attempts to isolate and remove the error-inducing fragment by dividing the input into progressively smaller parts. However, none of these parts individually cause the error, such that removing any one of them eliminates it.

This results in the solution `123456` with significant data loss, including the loss of structure and change in input fragment type from string to number. Hence, *DDMax* cannot effectively repair such inputs with multiple faults.

C. Effect of fragment decomposition

DDMax can produce non-optimal results even when the errors are contiguous, and hence considered *single* by *DDMax*. The problem happens when the corruption in the input interacts with the fragment decomposition algorithm of *DDMax*. As an example, consider a variant of the previous input: `[*+]`. The JSON string is invalid here because it contains two invalid characters which are contiguous. The operation of *DDMax* (Figure 1) is as follows:

1) The operation starts with $\mathit{DDMax}_2(\emptyset, 2)$
2) $|c_\times - \emptyset| \neq 1$. Hence, the base case does not apply
3) Can we increase to complement?
   $c_\times - \Delta_1$= `+]` ×
   $c_\times - \Delta_2$= `[*` ×
4) Can we increase to subset?
   $\emptyset \cup \Delta_1$=`[*` ×
   $\emptyset \cup \Delta_2$= `+]` ×
5) Can we increase granularity? $n < |c_\times - c'_\checkmark|$
   which is $2 < |c_\times - \emptyset|$ ✓
   Hence the next iteration is: $\mathit{DDMax}_2\big(\emptyset, 4\big)$
6) $|c_\times - \emptyset| \neq 1$. Hence, the base case does not apply.
7) Can we increase to complement?
   $c_\times - \Delta_1$= `*+]` ×
   $c_\times - \Delta_2$= `[+]` ×
   $c_\times - \Delta_3$= `[*]` ×
   $c_\times - \Delta_4$= `[*+` ×
8) Can we increase to subset?
   $\emptyset \cup \Delta_1$=`[` ×
   $\emptyset \cup \Delta_2$=`*` ×
   $\emptyset \cup \Delta_3$=`+` ×
   $\emptyset \cup \Delta_4$=`]` ×
9) Can we increase granularity? $4 < 4$ ×
10) The solution is $\emptyset$.

That is, *DDMax* is unable to repair this invalid JSON string. To understand its impact, consider `{ "item": "Apple", "price": ***3.45 }` in Table I which is from Kirschner et al. [19, Fig. 1] but with an extra `*`. The *DDMax* repair is `   3.45`, resulting in loss of structure and data.

The issue arises from *DDMax*'s partitioning strategy, which fails to isolate the error-causing fragment, even when it is contiguous. Despite the error being localized, no single removable fragment is identified that eliminates the error. Consequently, *DDMax* continues to search for increasingly smaller fragments, inadvertently discarding larger portions of potentially valid data in the process.

D. Limited Repertoire for Repair

Another major limitation of *DDMax* is that the only operation in its toolbox is *deletion* of tokens. Consider the following fragment: `{ "name": "Dave" "age": 42 }`. Here, there is a missing comma. The *DDMax* repair of this string will result in just `   42`, which is an *unexpected and significant reduction* from the original string. The issue is that, deletion of fragments alone can lead to significant corruption of information. In this instance, the availability of *insertion* could have repaired the input string to `{ "name": "Dave", "age": 42 }`. As *DDMax* is unable to insert tokens, such opportunities can be missed.

E. Single repair-candidate

Consider the following corrupted data `["A",[1,2]"]`. Given *deletion* as the only option, there are two possible repair candidates here: (1) `["A,[1,2]"]` and (2) `["A",[1,2]]`. Either may be the correct one. However, *DDMax* has to choose just one candidate, and the choice depends exclusively on which fragment was tested first. This means that *DDMax* cannot rely on a post-processing intelligent selector such as an end-user or an automatic validator even if one is provided.

F. Requirement of Valid Intermediates (Waypoints)

A severe limitation of *DDMax* is the assumption of a *valid passing subsequence* that can be progressively *extended* to form the final solution that is closest to the original record [19, Sec. 3.2]. This particular assumption may not be warranted in many real-world scenarios. For example, consider a parser for date-time. It expects strings that follow the format `YYYY-MM-DDTHH:MM:SSZ`. Given corrupt inputs such as `X2024-07-23T12:34:56Z`, `1925X-09-13T10:14:16Z`, and `1999-12-20T12:34:56ZX`, there is no *empty date* that *DDMax* can start from, that is applicable to all inputs even though the repair only requires `X` to be deleted.

G. Susceptibility to Local Maxima

DDMax assumes that a valid passing minimal subsequence can be progressively extended to the maximal valid subsequence. However, even when such a minimal subsequence exists, extending it maximally may not result in greater data retention compared to the original (corrupted) string.

For example, consider the corrupt JSON input string `{ "name": "Dave" "age": 42 }`. *DDMax* starts by identifying a minimal passing subsequence—the empty sequence. It then extends this sequence with `4`, and continues to extend it to `   42`. However, at this point, no additional fragments from the original input can be added without violating the parser constraints. As a result, potentially recoverable information is discarded because initial repair decisions constrain future possibilities, demonstrating *DDMax*'s susceptibility to local maxima.

III. Repairing Data with ϵRepair

Our approach was motivated by the following observation. Consider this output from `jq`, a popular `JSON` processor.

```
$ echo -n '{"name": "Dave" "age":42}' | jq .
parse error: Expected separator at line 1,
column 21
```

On providing an input that contains the missing comma, the parser responds with an *approximate* location of the error. If we instead provide `jq` with a truncated input, the parser responds with `EOF` indicating incomplete input,

```
$ echo -n '{"name": "Dave" ' | jq .
parse error: Unfinished JSON term at EOF at
line 1, column 16
```

requiring further data for completion. This behavior (i.e., detecting a *viable-prefix*) is supported by a wide variety of parsers including formal general-purpose parsers [32]–[34]. Our insight is that we can leverage this behavior to our advantage and attempt to repair the input based on a *minimal reliance* on the parser feedback. In particular, our approach ϵRepair only requires that the parser is able to distinguish between *incorrect* input and *incomplete* input. Given a parser that obeys this minimal contract, we can quickly determine the location of the error, and identify the *minimal* edits required to fix the parser error[2].

The following terms are used in our definition of ϵRepair:

alphabet The set of characters.

string A sequence of elements of an alphabet.

edit An edit is deletion or insertion of a single character.

parser An input processor that reads a string and indicates whether the string is *accepted* (✓), or *rejected* (×).

conforming parser A *parser* that instead of just rejecting the string, also specifies whether the string is *incorrect* (⋫) or merely *incomplete* (⊳), that is, a suffix exists that will make the complete string acceptable to the parser.

viable-prefix A string that when passed to the conforming parser, results in ⊳ or ✓ response from the parser.

parse-boundary Length of the maximal viable-prefix of a string according to a conforming parser. To the right of the parse-boundary is the maximal *viable-prefix* and to the left is the *remaining-suffix*.

[2]Conversely, error location, if available, can be leveraged for the same contract.

Algorithm 1 Boundary search for ϵREPAIR

```
function BSEARCH(string, parser)
    left, right ← 0, LEN(string)
    if PARSER(string[:right]) ∈ {▷, ✓} then
        return right
    while left < right − 1 do
        middle ← (left + right) // 2
        if PARSER(string[:middle]) ∈ {▷, ✓} then
            left ← middle
        else
            right ← middle
    return left
```

repair An *edit* made to the string that extends its *parse-boundary* or reduces the *remaining-suffix*.

repair-thread A sequence of *repairs* made on an input string. It has a single parse-boundary—the parse-boundary of the last repair in this thread, a corresponding *viable-prefix*, and the *remaining-suffix*.

repair-candidate A repair thread becomes a *repair-candidate* when the repaired string elicits the ✓ response from the parser, and there is no remaining-suffix.

edit-distance The number of minimal edits required to transform one to the other.

repair-distance The edit-distance from corrupt string to the repair-candidate.

thread-queue A priority queue of *repair-threads* sorted by the number of repairs and the parse-boundary.

The problem of *data repair* is: Given a *string* and a *parser*, find the least number of *repairs* to make the string pass the *parser*. Using a *conforming parser*, the ϵREPAIR algorithm can repair the string with the following steps:

1) *Boundary search.* Given a corrupt string, ϵREPAIR starts a search for the parse-boundary (Algorithm 1). This is then used to construct the *thread-queue* containing a single empty repair-thread, with the parse-boundary set to the search result.
2) *Apply Repair.* Starting with any existing repair-thread, ϵREPAIR applies either a deletion or an insertion on the *remaining-suffix*. This results in a new repair-thread.
   - An *insertion* inserts a character to the beginning of the *remaining-suffix*. Let $S^{'}$ be the viable-prefix of the current repair-thread, and c be the character. We create one repair-thread for each c in the alphabet where the parser responds with *incomplete* for $S^{'}+c$, with the new viable-prefix $S^{'}+c$.
3) *Extend Boundary.* For each repair-thread that results from the previous step, we attempt to extend the parse-boundary by adding additional characters from the remaining-suffix.
4) *Check Candidates.* If any thread results in a *repair-candidate*, then we return the candidate.
5) *Select Threads.* The algorithm selects the repair-thread from the priority queue which has recovered the maximum data so far from the original string, and continues through step 2. Duplicate threads (those that result in the same string) are discarded, keeping only those with the minimal number of repairs. A response of ✓ is treated the same as ▷ if the remaining-suffix is non-empty.

Algorithm 2 ϵREPAIR Algorithm

```
function DREPAIR(string, parser)
    boundary ← BSEARCH(string, parser)
    pq ← PRIORITYQUEUE
    ADD(pq, (string, boundary, ∅))
    while pq ≠ ∅ do
        (string, boundary, fixes) ← TOP(pq)
        deletes ← {(boundary, ∅)}
        inserts ← {(boundary, c) | c ∈ α,
          PARSER(string[:boundary]+c) ∈ { ▷, ✓ }}
        for fix in deletes ∪ inserts do
            new_str ← APPLYFIXES(string, fixes + [fix])
            new_boundary ← BSEARCH(new_str, parser)
            if PARSER(new_str[:new_boundary]) = ✓ then
                if string[new_boundary:] = ∅ then
                    return new_str
            ADD(pq, (string, new_boundary, fixes + [fix]))
    return ∅
```

The algorithm (Algorithm 2) returns as soon as the first repair-candidate is found. However, one may also repeatedly invoke the algorithm to produce an ordered ranking of further repair-candidates.

A. ϵREPAIR *in action*

As an example, let us consider the following input to the JSON processor from Table I: `{ "name": "Dave" "age": 42 }`

1) ϵREPAIR starts by executing a binary search for the boundary where the input prefix changes from ▷ to ⊬. The boundary is obtained at index 17, providing the viable-prefix `{ "name": "Dave"` and the remaining-suffix `"age": 42 }`
2) There are three possibilities for repair here. The first is to delete the next character `"` from the remaining-suffix. This however, does not change the parse-boundary, as the string `{ "name": "Dave" a` still results in ⊬ response from the JSON processor. The second is to insert a character. We only insert characters that will lead to an improvement in the parse-boundary. Here, the only possibility is `,`, resulting in the string `{ "name": "Dave" ,`.
3) Extending the parse-boundary by appending remaining characters in the remaining-suffix, we have `{ "name": "Dave" ,"age": 42 }`, and the response ✓.

Let us consider a second example, again from Table I: `{"ABCD":[*"1,2,3,4,5,6"]*}`. We follow a single repair thread for ease of explanation.

1) ϵREPAIR starts by finding the parse-boundary. This is obtained at index 9, providing the viable-prefix `{"ABCD":[` and remaining-suffix `*"1,2,3,4,5,6"]*}`.

2) ϵREPAIR then appends the next character `*` to the input, resulting in `{"ABCD":[*` and observes the result. In this case, the JSON processor returns ↯.
3) Hence, the newly added character is discarded, and the character at the next index is appended, resulting in `{"ABCD":["`. JSON processor responds with ▷.
4) ϵREPAIR now appends the character in the next index, forming `{"ABCD":["1` and ▷ from the JSON processor.
5) Proceeding in this fashion, the input reaches `{"ABCD":["1,2,3,4,5,6"]*` at which point, we again have the response ↯ from the JSON processor. Hence, we discard this character, and try the next character, resulting in `{"ABCD":["1,2,3,4,5,6"]}`.
6) The JSON processor responds with ✓.

Completing and demonstrating multi-fault repair by ϵREPAIR.

ϵREPAIR Limitations. Let us now examine two cases where ϵREPAIR has counter-intuitive behavior. Consider the input `{ "item": "Apple", "price": ***3.45 }`.

1) As usual, ϵREPAIR starts by finding the parse-boundary, and the viable-prefix which is `{"item": "Apple", "price":` , followed by the remaining-suffix `***3.45 }`.
2) Since `*` is not a valid character to add, ϵREPAIR deletes this character. Deletion of the two remaining characters will result in `{"item": "Apple", "price": 3.45 }`, which is accepted by the JSON processor.
3) However, we also have possible characters that can be inserted at the first parse-boundary, which results in a parse-boundary extension. That is `"` will extend the parse-boundary to `{"item": "Apple", "price": "***3.45 }`. On continuation, ϵREPAIR finds that the following result with three insertions also results in an accepted string. `{"item": "Apple", "price": "***3.45 }"}`.

That is, there may be multiple repair candidates, and as ϵREPAIR operates without human intervention, it is unable to distinguish the semantics, affecting optimality of repair.

Another example is the input `"abcd":[1,2,3]}`.

1) ϵREPAIR starts by finding the parse-boundary and the corresponding viable-prefix, which is `"abcd"`.
2) The possible extensions are to delete `:` or to insert one of the characters at this point. Unfortunately, none of the characters in the alphabet can increase the parse-boundary. Hence, the only option that is possible is to continue to delete characters from the remaining-suffix. This leads to the string `"abcd"`, which is suboptimal.

That is, ϵREPAIR can also result in suboptimal repairs.

A final example is the input `"[1,2,3,4]`.

1) ϵREPAIR starts by finding the parse-boundary and the corresponding viable-prefix, which is `"[1,2,3,4]`.
2) The possible extension here is to append a character `"`, leading to the string `"[1,2,3,4]"`.

That is, it ignores the possibility of deletion of the first character as the correction even though that can produce to an optimal repair-candidate. We next discuss a mitigation strategy trading performance for accuracy.

Extended ϵREPAIR. When ϵREPAIR obtains an ▷ or ✓ response from the parser, it assumes that there can be no repairs in the viable-prefix thus obtained. However, as we saw, this need not be the case. For example, given `"abcd":[1,2,3]}`, the parser returns ✓ for the string `"abcd"`. However, as we found, a repair inserting `{` in the string beginning, resulting in `{"abcd":[1,2,3]}` can lead to more data recovery. To allow such repairs to take place, we extend the ϵREPAIR potential candidates as follows.

When ϵREPAIR obtains an ↯ response from the parser, we create repair-candidates not only at the end of the viable-prefix, but also at all points in the viable-prefix. That is, given the corrupt string, `"abcd":[1,2,3]}`, when we obtain ↯ with `"abcd":`, all the following locations indicated by red cursors are candidate locations for inserts; 1) `"abcd":` 2) `"abcd":` 3) `"abcd":` 4) `"abcd":` 5) `"abcd":` 6) `"abcd":` 7) `"abcd":`. Similarly, each character next to the red cursor becomes a potential candidate for deletion. The problem with this approach is that it may lead to numerous repair-candidates, with each location potentially having multiple candidates. That is, if we have n characters in the language, then each location can at worst produce $n+1$ candidates, and a viable-prefix of length l can result in $l \times (n+1)$ candidates. Hence, heuristic strategies are needed to reduce the number of repair-candidates. Such heuristics may be program or data specific. In our current empirical evaluation, we only use the simple ϵREPAIR, with no additional extensions. However, additional heuristics are one of the future research directions.

Post-processing. Another strategy is post-processing. ϵREPAIR can generate multiple repair-candidates, only one of which may be acceptable. For example, while ϵREPAIR may generate both `{"item": "Apple", "price": "***3.45 }"}` as well as `{"item": "Apple", "price": 3.45 }` (both 3 edits from the corrupt string), only one may be correct. Hence, we provide a ranked list of repair-candidates (ordered by repair-distance) to the user along with the details of repair. Users can do their own post-processing to identify the best candidate if needed. This is in line with the standard industry practice used by error-correcting parsers [23]–[25].

Similarly, ϵREPAIR can synthesize tokens that let the parse continue, and produce valid records. However, tokens thus produced do not hold semantic information. Hence, during post-processing, we expect to replace synthesized data with semantic content with placeholders as provided by the user.

IV. EVALUATION

How does ϵREPAIR compare with state-of-the-art? We propose the following research questions which are designed to investigate the effectiveness of ϵREPAIR in several dimensions.

A. Research Questions

We aim to evaluate the effectiveness of ϵREPAIR in comparison to existing techniques. While several input rectification

approaches rely on learning input features from examples or utilize input specifications, few focus exclusively on parser-based strategies. Among parser-centric, format-agnostic repair techniques, *DDMax* is the primary competitor to ϵREPAIR. However, the two approaches have differing requirements: *DDMax* depends on the ability to delete input fragments effectively, whereas ϵREPAIR requires accurate error feedback from the parser. As a result, although there is significant overlap in the types of inputs they can handle, not all subjects repairable by ϵREPAIR can be repaired by *DDMax*, and vice versa. Notably, the subjects used in Kirschner et al. [19] lack precise error feedback, rendering them unsuitable for direct repair using ϵREPAIR. Hence, we use a set of real-world parsers for evaluation that allows applying both *DDMax* as well as ϵREPAIR.

The input languages of our subject programs are also available in ANTLR format, and ANTLR provides the error-feedback that is required by ϵREPAIR. This allows us to ask: How does ϵREPAIR compare against format-dependent techniques such as *DDmaxG* and ANTLR? Hence, for the following RQs, we compare the effectiveness of ϵREPAIR against (a) *DDMax* and (b) *DDmaxG* and (c) ANTLR.

One of the most important questions regarding repair is the *quality of repair*. A repair is not useful if it results in discarding most of the data in the original record. Hence, we ask:

RQ1: What is the quality of data repair by ϵREPAIR in comparison to its competitors?

The second question we ask is whether the data can be recovered at all. That is, can the repair produce valid candidates:

RQ2: How many corrupt records can be repaired by ϵREPAIR in comparison to its competitors?

A third question that is relevant in this space is efficiency. That is, while it is expected that a more intelligent algorithm may take more runtime, it should be within practical bounds. Hence, the third research question is:

RQ3: How does ϵREPAIR compare to *DDMax* in performance? We next discuss our evaluation strategy.

B. Subject Programs

TABLE II: Subject programs used in the evaluation

	LOC	Parser Lang.	Input Format	Development
`ini`	511	C	`INI`	2009-2022
`cjson`	3413	C	`JSON`	2009-2022
`sexp`	978	C	`SExp`	2016-2016
`tinyc`	421	C	`TinyC`	2011-2018

TABLE III: Number of corrupt inputs

	Record Len.	Single Corr.	Double Corr.	Truncated
`INI`	102.0 ± 20.4	1000	100	100 (29.1%)
`JSON`	146.6 ± 46.6	1000	100	100 (26.7%)
`SExp`	66.8 ± 31.2	1000	100	100 (26.8%)
`TinyC`	45.3 ± 20.4	1000	100	100 (24.8%)

Average truncated suffix length is indicated with parentheses.

To enable accurate assessment, we investigated the effectiveness of ϵREPAIR on several real-world parsers such as `ini` (`INI`), `cjson` (`JSON`), `sexp` (`SExp`), and `tinyc` (`TinyC`). Details are given in Table II. These were first compared against the format-agnostic competitor *DDMax*. Each parser is moderately large (between 500 LOC to 3500 LOC), relatively mature (7 to 14 years of development), and written in C.

Next, to understand how ϵREPAIR performs against error recovery from general parsers, and data repair using format-dependent techniques, we investigated the effectiveness of ϵREPAIR against the format-dependent techniques such as *DDmaxG* and ANTLR on the same subjects.

Finally, regular expression matchers such as PCRE provide the parse-error feedback ϵREPAIR needs for repair. While the formal format is technically available, regular expression matchers do not provide the *parse tree* that *DDmaxG* requires, nor is the grammar available to be used with ANTLR. The format-agnostic *DDMax* is unusable against regular expressions because they do not provide a successful waypoint that is required by *DDMax*. **Hence, as ϵREPAIR is the only technique able to repair data that is described by regular expressions**, we investigate and benchmark the effectiveness of ϵREPAIR for several common data formats that are described by regular expressions (taken from RegExLib).

C. Test Data

We initially considered using real-world corrupt data, similar to the *DDMax* evaluation [19]. However, a key limitation of such data is the absence of a reliable *ground truth*. To verify the success of a repair, we must compare the repaired content to the *original content*—merely passing the parser without errors is insufficient. In particular, we aim to distinguish between simply deleting parts of the original string to eliminate corruption and actually replacing corrupt data with valid content. Without access to the original data, this distinction is impossible to make.

Therefore, we opted to use valid records for each input format under test, intentionally corrupt these records, apply repair algorithms, and then compare the results to the originals. Using Github API, we collected 450 `INI` files, 666 `JSON` files, and 820 `SExp` files. For `tinyc`, due to its limited adoption and the scarcity of real-world data, we generated complex strings conforming to the `TinyC` grammar using random generation.

We sampled 100 *valid records* for each format. Further details are provided in the *Record Len.* column of Table III.

Single corruption. First, we evaluated the effectiveness of ϵREPAIR in fixing simple corruptions. For each of the records, we induced a single character corruption by either (1) deleting, (2) substituting, or (3) inserting a single character. If the corruption did not result in a parse error, another corruption was induced on the original string as a replacement, and the process continued until we had a corrupt record with a single corruption. For each input record, this process was repeated ten times, resulting in a total of $10 \times 100 = 1000$ corrupt records per format (column *Single Corr.* in Table III).

Double corruption. To simulate more complex, real-world scenarios where multiple consecutive locations are affected,

we induced corruptions involving two consecutive characters. These are summarized under *Double Corr.* in Table III.

One of the challenges in data repair we observed is that its effectiveness diminishes as the number of corruptions increases. Each additional corruption makes repair computationally expensive and increases the risk of altering structural information in ways that impair accurate reconstruction, especially on the interpretation of the remaining data. Consequently, we limited our experiments to cases involving up to two random corruptions.

Truncations. An exception to the above limitation is truncation, where only a prefix of the original record is preserved. In such cases, the interpretation of the truncated data remains straightforward because structural anomalies introduced in the suffix do not affect the surviving data. Truncation is common during interrupted network transmission, where only partial data may be transferred. To simulate this, we truncated each record at a random index after the midpoint and evaluated ϵREPAIR 's ability to reconstruct a valid record from the preserved prefix. *Truncation* in Table III has the details. The average length of data deleted is provided in parentheses.

D. Baseline Comparisons

We compare ϵREPAIR to the following repair techniques:

Format-free. As ϵREPAIR does not require the format specification to repair data, we need to compare it to similar techniques that do not require a format specification.

(1) Lexical *DDMax*: Lexical *DDMax* by Kirschner et al. [19] is the only option for format-free data repair. It is labeled as *DDMax* in all tables.

Format-dependent. While ϵREPAIR is designed to be used when the formal data-specification is not available, it can also be used to repair data when the data-specification is available by leveraging the general parser feedback. Hence, we compare it to the following format-dependent techniques.

(2) Syntactic *DDMax*: The *DDMax* based on grammar (represented as *DDmaxG* in all tables) is a counterpart to Lexical *DDMax* that uses the grammar specification to parse the incoming data, and leverages the *parse tree* thus obtained for data repair [19].

(3) ANTLR: As ANTLR provides its own error repair, we also use ANTLR as a baseline.

E. Platform

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

F. Research Protocol

We ran each repair technique on each file and collected metrics. We set a time limit of four minutes per record for ϵREPAIR. The other techniques did not require a timeout.

TABLE IV: Distance between corrupt data and repaired data. The highlighted values indicate best repair.

	Subject	Format-free		Format-dependent	
		ϵREPAIR	***DDMax***	***DDmaxG***	**ANTLR**
Single	INI	**1.4** ± 0.8	2.5 ± 3.0	26.2 ± 6.4	25.0 ± 5.6
	JSON	**5.1** ± 19.0	26.0 ± 43.7	48.5 ± 31.5	40.4 ± 24.3
	SExp	10.3 ± 19.0	**7.6** ± 14.7	36.9 ± 27.2	N/A ± N/A
	TinyC	**4.1** ± 6.5	9.2 ± 13.5	25.1 ± 11.7	21.8 ± 10.4
Double	INI	**1.5** ± 0.9	3.0 ± 3.4	26.6 ± 6.8	24.8 ± 5.7
	JSON	**7.0** ± 24.3	43.0 ± 52.1	50.1 ± 32.6	40.0 ± 27.0
	SExp	12.0 ± 19.5	**10.9** ± 16.7	39.5 ± 27.3	N/A ± N/A
	TinyC	**6.6** ± 5.1	27.6 ± 15.4	27.3 ± 12.5	20.7 ± 10.8
Truncated	INI	**1.0** ± 0.0	2.0 ± 1.9	18.6 ± 5.0	17.8 ± 4.6
	JSON	**3.3** ± 1.5	74.3 ± 29.6	83.2 ± 34.6	N/A ± N/A
	SExp	**1.8** ± 0.4	22.2 ± 18.7	35.2 ± 22.1	N/A ± N/A
	TinyC	**1.9** ± 0.8	22.3 ± 9.0	28.0 ± 9.7	N/A ± N/A
Average		**5.1** ± 13.9	13.5 ± 27.3	35.0 ± 24.8	29.0 ± 17.4
Recovery		**94%± 0.2%**	83%± 0.3%	80%± 0.2%	91%± 0.1%

V. RESULTS

RQ1: What is the quality of data repair by ϵREPAIR and its competitors? To answer this question, we collected all corrupt data-records that could be successfully repaired by each technique. Next, for each such record pair, we computed the *edit-distance* between the corrupt record and the repaired record. This is given in Table IV. The standard deviation is given after the symbol ±. The best overall values are marked in bold. The overall average is given in the last row.

The data obtained indicates that ϵREPAIR, when it repairs records, *does so with the least amount of modifications*. This is true across all kinds of corruptions, and across all formats except in the case of SExp-single and SExp-double. In these two cases, there is very little difference between the mean values for ϵREPAIR and *DDMax* when considering the standard deviation. In all other cases, there is a stark difference in favor of ϵREPAIR.

Overall ϵREPAIR repairs required on average 5.1 edits to the given record compared to 13.5 for *DDMax*, and 35.0 for *DDmaxG*. The corrections made by ANTLR were on average 29.0 edits away.[3]

> *Overall, ϵREPAIR produces the best repairs, requiring only 5.1 edits of the corrupted record on average, which is 2.6 × better than the nearest competitor DDMax with 13.5 edits.*

Recovery of Data. A key concern when attempting data repair is how much of the original data can be recovered. That is, (1) how much of the original data needed to be thrown away as corrupt, and (2) how much of new data needed to be added to make the record valid. This can be empirically evaluated by tabulating the *delete* operations in computing the edit-distance.

In Table IV, the *Recovery* row shows how much of the data from the original string remains in the repaired string. Given *original string length* of L, and the *deleted data length* of d, this is computed as $mean(\frac{L-d}{L})$. Our results show that ϵREPAIR recovers 94% of the original data. In comparison, *DDMax* could only recover 83% of the original data. That

[3]Note that this is based on repaired inputs. ANTLR repaired noticeably fewer records compared to other techniques.

TABLE V: Distance from original data to repaired data

	Subject	Format-free		Format-dependent	
		ϵREPAIR	*DDmax*	*DDmaxG*	ANTLR
Single	INI	**2.4** ± 0.8	3.3 ± 2.7	27.3 ± 6.3	25.8 ± 5.6
	JSON	**5.3** ± 19.0	26.1 ± 43.6	48.5 ± 31.5	40.4 ± 24.3
	SExp	13.3 ± 19.2	**8.6** ± 14.5	37.4 ± 27.0	N/A ± N/A
	TinyC	**4.1** ± 6.5	9.2 ± 13.5	25.0 ± 11.8	21.4 ± 10.4
Double	INI	**3.5** ± 0.9	4.8 ± 3.0	28.2 ± 6.8	26.6 ± 5.7
	JSON	**7.8** ± 24.2	43.7 ± 51.6	50.7 ± 32.4	40.6 ± 27.0
	SExp	13.3 ± 19.3	**12.5** ± 16.3	39.3 ± 27.4	N/A ± N/A
	TinyC	**6.5** ± 5.2	27.6 ± 15.4	27.1 ± 12.5	20.4 ± 10.7
Truncated	INI	**28.1** ± 15.9	30.6 ± 16.2	45.5 ± 15.6	44.7 ± 15.0
	JSON	**35.1** ± 23.1	111.9 ± 35.3	121.8 ± 41.7	N/A ± N/A
	SExp	**15.7** ± 11.4	40.1 ± 24.0	51.1 ± 27.1	N/A ± N/A
	TinyC	**7.8** ± 7.6	36.6 ± 13.4	39.1 ± 11.7	N/A ± N/A
Average		**7.0** ± 15.4	16.0 ± 29.8	37.2 ± 27.2	30.4 ± 17.7
Recovery		**92%± 0.2%**	82%± 0.3%	79%± 0.2%	90%± 0.1%

TABLE VI: Number of Corrupt Records Repaired

	Subject	Format-free		Format-dependent	
		ϵREPAIR	*DDmax*	*DDmaxG*	ANTLR
Single	INI	**1000**	**1000**	**1000**	884
	JSON	**999**	971	982	703
	SExp	966	**1000**	**1000**	0
	TinyC	**1000**	984	984	481
Double	INI	**100**	**100**	**100**	91
	JSON	98	**99**	98	68
	SExp	94	**100**	**100**	0
	TinyC	**100**	98	98	28
Truncated	INI	**100**	**100**	**100**	B100
	JSON	82	90	**100**	1
	SExp	39	**100**	**100**	0
	TinyC	**82**	77	77	4
Total		4660	4719	4739	2355

TABLE VII: Number of *perfectly* repaired files

Subject	Format-free		Format-dependent	
	ϵREPAIR	*DDmax*	*DDmaxG*	ANTLR
INI	0	0	0	0
JSON	**25**	0	0	0
SExp	**7**	0	0	0
TinyC	**63**	0	0	0

TABLE VIII: Efficiency of data repair (Average)

Metric	Format-free		Format-dependent	
	ϵREPAIR	*DDmax*	*DDmaxG*	ANTLR
Runtime	3.87 secs	2.7 secs	2.0 secs	0.3 secs
#Execs	897	787	**569**	N/A

is, the repair by ϵREPAIR shows an improvement of 11% (or 2.8× reduction in data loss) over repair by *DDMax*.

Detailed Evaluation of Repair Quality. A corrupt data record has likely originated from an intact one, and the repair quality should be judged not just by the repair distance, but also by how close the repaired data are to the original data.

The edit-distance between intact records and their repairs from the competing techniques is given in Table V.

Our data shows that ϵREPAIR repairs are closest to the original data. This is true across all kinds of corruptions, and across all formats except for SExp single and double corruptions, and INI truncation. We also find that in these exceptional cases, the mean difference is very little between ϵREPAIR and *DDMax* when considering the standard deviation.

Overall ϵREPAIR produced repairs that were on average 7.0 edits away from the original record compared to 16.0 for DDMax, an improvement of 2.3×.

RQ2: How many corrupt records can be repaired by ϵREPAIR? This experiment compares the number of files repaired by ϵREPAIR with both format-free and format-dependent techniques. The results are tabulated in Table VI. The format-free and format-dependent techniques are separated by a partition. The overall best values are marked in bold type.

The results indicate that in format-free methods ϵREPAIR was able to repair slightly more files on single corruption records, while *DDMax* were able to repair slightly more double-corruption records. For truncation, *DDmaxG* managed to repair all records followed by *DDMax* and ϵREPAIR. In general, the performance of *DDmaxG* is as expected because *DDmaxG* operates with the knowledge of the input format. As Kirschner et al. [19] observe, ANTLR does not reliably perform data repair in any of the cases.

ϵREPAIR was able to repair 97% of all records, which is comparable to 98% from DDmaxG and DDmaxG.

Perfect repair. While data recovery metrics offer some insight into the quality of repairs, one key question remains: does the repair fully and accurately restore the corrupted data?

We evaluated the number of records perfectly repaired by each approach. That is, the repaired record is *identical* to the original before corruption. Our results show that only ϵREPAIR was able to achieve perfect repair for 95 records overall.

RQ3: How does ϵREPAIR compare to *DDMax* in performance?

The runtime of all three approaches is shown in Table VIII. The *Runtime* is *DDmaxG* is the fastest, with an average runtime of 2 seconds and requiring around 569 parser executions per repair. In comparison, ϵREPAIR is 1.9× times slower than syntactic *DDmaxG* and 1.4× slower than *DDMax*.

The primary reason for the increased runtime in ϵREPAIR is the additional insert repair operation, where each character in the alphabet needs to be checked. This process can be time-consuming, especially in formats with large alphabets. Despite being 40% slower, ϵREPAIR's average runtime of 3.9 seconds per record remains practical for data repair (cf. Table VIII).

Although ϵREPAIR is 40% slower than DDMax, its average runtime of 3.8 seconds per record is still practical for data repair.

Our evaluation shows that ϵREPAIR outperforms *DDMax* while relaxing the constraints placed on the parser.

A. Repairing regular data-formats

Data validation is commonly performed using regular expressions. Corruptions in records validated by regular expressions cannot be fixed by *DDMax*. For example, given the regular expression

TABLE IX: ϵREPAIR on regular expressions

Formats	Total	Success rate	Repair-Distance
filepath	100	100%	0.96 (0.2)
date	100	100%	1.13 (0.7)
ipv6	100	100%	0.88 (0.4)
time	100	100%	0.90 (0.7)
url	100	100%	0.45 (0.5)
ipv4	100	100%	0.91 (0.5)
isbn	100	100%	1.67 (0.5)

`[0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]` and a non-conforming record `24-02-01` there is no successful empty record for *DDMax* to start from, nor are there any valid waypoints.

We investigated the effectiveness of ϵREPAIR against such records, given in Table IX. Indeed, ϵREPAIR was able to repair all corruptions, and we note that **ϵREPAIR is the only technique that can repair such records** (i.e., no benchmark to compare against).

We also noted that the data formats in Kirschner et al. [19] were available as ANTLR specifications (with some variations). We further noted that the ANTLR parser can provide the feedback required by ϵREPAIR. Hence, we evaluated ϵREPAIR and competitors against these ANTLR parsers. The results are provided in the Appendix. The results broadly indicate the superiority of ϵREPAIR in data repair.

VI. DISCUSSION

Our evaluation shows that ϵREPAIR consistently outperforms *DDMax* and its format-dependent variant *DDmaxG* in data repair. Our innovations are (1) the significantly relaxed constraint in the subject parsers (avoiding valid waypoints, and instead requiring valid parser feedback), (2) incorporating a larger repertoire of repairs, and (3) while incurring a performance penalty due to the increased intelligence, remaining practical to use in large data dumps.

The ϵREPAIR algorithm can be applied for data repair in any of the following circumstances:

1) You have a parser that provides reasonable feedback when it encounters a parse error. Such parsers are common in software engineering including standard handwritten parsers, as well as regular expression matchers.
2) The parser can be instrumented to provide conforming error-feedback which is often used in fuzzing [35].
3) A formal grammar or a regular expression is available to validate the input, in which case, any of the general context-free parsers can be modified to provide the required error feedback, or a common library such as PCRE can be used with the partial match feature (`PCRE_PARTIAL`). Note that in comparison to *DDmaxG* and ANTLR, ϵREPAIR only requires the general CFG parser to provide feedback, and does not require a parse tree to be constructed.

If any of these circumstances apply, then ϵREPAIR is an optimal algorithm that can produce an optimal repair.

During our evaluation, we find that ϵREPAIR is able to produce consistently high quality repairs, and are able to produce repairs that are closest to the *original data before corruption*, an improvement of 2.3× over *DDMax* (c.f. Table V). Indeed, ϵREPAIR is the only technique that achieved *perfect repairs* achieved in several corrupted records (c.f Table VII).

A. Limitations of ϵREPAIR

While ϵREPAIR is an improvement over the state of the art, it still has several limitations, which we discuss next.

Limitations due to closing. While ϵREPAIR produces better repairs than its competitors in most cases, ϵREPAIR is less effective than *DDMax* in some, especially when the input format is highly recursive. This is what happens for SExp. In this case, given an input such as `(1+` ϵREPAIR needs to find the closing `)` to continue and to close the record. The issue here is that the character `(` is also a continuation, and if this path is taken `(1+(` requires two closing `)`. As no character is privileged (we do not assume anything about the alphabet of the input format), closing the input can get progressively difficult, which is reflected in Table VI.

Limited data repair. As with *DDMax*, ϵREPAIR only attempts to recover as much data as possible from a corrupt record. This means that there is a chance of spurious interpretation of existing data (in the case of *DDMax*) and also incorporating spurious data (in the case of ϵREPAIR, ANTLR and other error-correcting parsers [23]–[25]), either of which may have unintended semantic consequences. While this can be worked around with a post processing step and user supplied validators, there is no guarantee that the end result is exactly the same as the original. That is, both *DDMax* and ϵREPAIR run the risk of further data-corruption and information distortion, and the end-users should select the best semantically-fit repair from the repair-candidates provided by ϵREPAIR.

Requirement for robust conforming parsers. When the parser is non-conforming (i.e., does not provide ▷ and ↯ feedback), ϵREPAIR is unable to use the parser feedback guidance. This reduces the quality of repair drastically. However, in such cases, ϵREPAIR can still repair those parts of the format that are conforming to the feedback requirements.

Inability to work with context-sensitive constraints. Some formats such as binary files require the format specification to be context sensitive. That is, they require the content-length to be specified before the content. For such formats, ϵREPAIR is of limited use. Furthermore, some formats may require checksums or hash values. Repairing such extra constraints that are beyond context-free is impossible with ϵREPAIR.

Corruptions that do not introduce immediate parse-errors. Some data corruptions do not introduce parse-error immediately. An example is the JSON string `[[[""]]]`, where the `"` starts a string literal that simply continues accepting the remaining characters. We note that, while this is a limitation, this can be worked around either by adopting more extended repairs or by limiting ϵREPAIR repairs to deletions if this behavior is expected.

B. Applications of ϵREPAIR

Beyond large-scale data repair, there are a few more areas ϵREPAIR could prove beneficial. These include:

- Interactive form validation: If form validation is accomplished by regular expressions, ϵREPAIR could suggest a ranked list of minimal edits to users.
- Autocomplete: ϵREPAIR could perform well on IDE auto-completion tasks, where the task is to provide an autocomplete that best fits the current context.
- Compilers: ϵREPAIR could also offer principled code fixes for syntax errors.

VII. Related Work

A. Constraint-based Input Repair

These methods learn constraints from input data [36], [37] or use specified constraints [38]–[40] to guide repair. The difference from ϵREPAIR is that these constraints are learned from sample inputs while with ϵREPAIR, the inputs are rectified using the parser feedback.

B. Black-box Input Repair

DDMax [19] was the first black-box technique focused on maximizing repair via deletion. ϵREPAIR builds on this by adding insertion and using different parser feedback.

C. White and Gray-box Input Repair

Techniques like *docovery* [41] use symbolic execution, while others analyze fault regions [42]. These require program analysis, unlike the black-box ϵREPAIR.

D. Parser-directed Input Repair

Compilers/parsers often include error recovery [23]–[25], [43], [44] using heuristics like symbol insertion/deletion/replacement [45]–[47], forward/backward moves [48], [49], or panic mode. These often require grammars (ANTLR [25]) and prioritize continued parsing over optimal data recovery. ϵREPAIR aims for minimal-edit recovery without necessarily needing a grammar (if the parser provides feedback).

E. LLM-based Input Repair

LLM-based input repair [50]–[52] is likely to perform well on common formats like `INI`, `JSON`, and `SExp`, especially when examples of such data formats appear in training corpora. However, our focus remains on algorithmic improvements for several reasons. Algorithms can be invoked by LLMs as tools, do not hallucinate, and are unaffected by model choice, prompt design, or context length. They also offer advantages in correctness guarantees, efficiency, transparency, and generalization—making them reliable both as standalone methods and as components within LLM-based pipelines. We also note that it is also hard to find programs that are not already in the training data for evaluation.

VIII. Threats to Validity

External Validity. Our evaluation considered four common data formats. However, we note that all our parsers are implemented in C, and were chosen for the relatively precise error feedback. This may not generalize well to parsers in other programming languages or to parsers that provide coarser or incorrect error-feedback, which can have an impact on the repair quality. Furthermore, our evaluation focuses on character-level edits based on a known alphabet–ASCII. Real-world systems may require Unicode, or higher-level token or semantic edits, which are not captured in our current model. Additionally, while we simulate a range of corruption types, our corruption model may not fully reflect real-world corruption patterns, which can be clustered or domain-specific.

Internal Validity. Our implementation correctness and baseline reimplementations are potential sources of bias. We mitigated this by testing against known cases and using public or carefully reimplemented baselines, but undetected issues may remain. All experiments were conducted on high-performance Apple M2 Ultra hardware; performance may degrade on commodity or resource-constrained systems. We selected a fixed four-minute timeout for repairs without sensitivity analysis. It is possible that different time budgets would yield different repair quality or coverage. We also terminate search after finding the first valid repair, which may lead to suboptimal results due to early search commitment.

Construct Validity. We evaluate repair quality using edit distance, recovery percentage, and parser acceptance. These measures capture structural similarity but not semantic correctness or fitness for downstream tasks. We do not include human or domain-expert validation of repair utility, which limits the interpretability of our results in practical settings. While we generate ranked lists of repair candidates, we do not quantify the risk of suboptimal candidates ranking higher than semantically correct ones. Our method depends on parser feedback quality; we do not evaluate how imprecise or misleading feedback affects repair outcomes. Future work should explore ranking quality, human validation, and sensitivity to parser feedback reliability to provide a more complete assessment.

IX. Conclusion

We present ϵREPAIR, a novel format-free data repair approach that leverages parser feedback ($\triangleright$ vs. $\not\triangleright$) to guide repair through insertions and deletions. It relaxes the constraints of *DDMax* (no need for valid empty states or waypoints) and expands the repair capabilities.

Our evaluation demonstrated that ϵREPAIR achieves significantly higher-quality repairs than *DDMax* (2.6× better corrupt-to-repair distance, 2.3× better original-to-repair distance) and reduces data loss by 2.8×, with only a 1.4× runtime overhead. It uniquely achieved perfect repairs in 95 cases and successfully repaired data validated by regular expressions, where *DDMax* fails. ϵREPAIR offers a practical, robust, and more effective alternative for automatic data repair when format specifications are missing or unreliable.

Our implementation, data and results are available here:

https://github.com/Jacksadventure/epsilonrepair-artifact

References

[1] C. C. Wood and W. W. Banks, "Human error: an overlooked but significant information security problem," *Computers & Security*, vol. 12, no. 1, 1993.

[2] I. C. Marcin Kozak, Wojtek Krzanowski and J. Hartley, "The effects of data input errors on subsequent statistical inference," *Journal of Applied Statistics*, vol. 42, no. 9, pp. 2030–2037, 2015.

[3] S. G. Powell, K. R. Baker, and B. Lawson, "A critical review of the literature on spreadsheet errors," *Decision Support Systems*, vol. 46, no. 1, pp. 128–138, 2008.

[4] M. Woodard, S. Sedigh Sarvestani, and A. Hurson, "A survey of research on data corruption in cyber-physical critical infrastructure systems," *Advances in Computers*, vol. 98, pp. 59–87, 12 2015.

[5] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Workshop on Hot Topics in Operating Systems*, 2021, pp. 9–16.

[6] C. Scaffidi and M. Shaw, "Accommodating data heterogeneity in uls systems," in *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, 2008, pp. 15–18.

[7] D. Core, "Combatting data corruption in the digital age," https://www.datacore.com/glossary/data-corruption/, 2021.

[8] K. Muşlu, Y. Brun, and A. Meliou, "Preventing data errors with continuous testing," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 373–384.

[9] C. Scaffidi, M. Shaw, and B. Myers, "Estimating the numbers of end users and end user programmers," in *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 2005, pp. 207–214.

[10] M. Fisher and G. Rothermel, "The euses spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms," in *Workshop on End-user software engineering*, 2005.

[11] R. R. Panko, "What we know about spreadsheet errors," *JOEUC*, vol. 10, no. 2, pp. 15–21, 1998.

[12] N. Harrand, T. Durieux, D. Broman, and B. Baudry, "The behavioral diversity of java json libraries," 2021.

[13] N. Seriot, "Parsing json is a minefield," https://seriot.ch/projects/parsing_json.html, 2016.

[14] T. Arvin, "Comparison of different sql implementations," https://troels.arvin.dk/db/rdbms/, 2018.

[15] E. S. Raymond, "The Art of Unix Programming, Chapter 5. Textuality," http://www.catb.org/~esr/writings/taoup/html/ch05s02.html, 2003.

[16] M. Jimoh, *Date Formats: A Rare Headache in the Construction of Contractual Documents*. SSRN, 2023.

[17] D. Brownell, "XML Conformance Update," https://www.xml.com/pub/2000/05/10/conformance/conformance.html, 2000.

[18] F. Ridzuan and W. M. N. Wan Zainon, "A review on data cleansing methods for big data," *Procedia Computer Science*, vol. 161, pp. 731–738, 2019, the Fifth Information Systems International Conference, 23-24 July 2019, Surabaya, Indonesia. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050919318885

[19] L. Kirschner, E. Soremekun, and A. Zeller, "Debugging inputs," in *ICSE*, 2020, p. 75–86.

[20] C. Scaffidi, B. Myers, and M. Shaw, "Topes," in *ICSE*. IEEE, 2008, pp. 1–10.

[21] M. A. Hernández and S. J. Stolfo, "Real-world data is dirty: Data cleansing and the merge/purge problem," *Data mining and knowledge discovery*, vol. 2, pp. 9–37, 1998.

[22] M. Musleh, M. Ouzzani, N. Tang, and A. Doan, "Coclean: Collaborative data cleaning," in *International Conference on Management of Data*, ser. SIGMOD '20, 2020, p. 2757–2760.

[23] A. V. Aho and T. G. Peterson, "A minimum distance error-correcting parser for context-free languages," *SIAM Journal on Computing*, vol. 1, no. 4, pp. 305–312, 1972.

[24] L. Diekmann and L. Tratt, "Don't panic! better, fewer, syntax errors for LR parsers," in *ECOOP*, ser. LIPIcs, R. Hirschfeld and T. Pape, Eds., vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 6:1–6:32.

[25] T. Parr and K. Fisher, "Ll (*) the foundation of the antlr parser generator," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 425–436, 2011.

[26] WHATWG, "Url living standard," https://url.spec.whatwg.org/, 2022.

[27] Z. Sims, https://www.canva.dev/blog/engineering/when-url-parsers-disagree-cve-2023-38633/, 2023.

[28] J. Gruber, "Introducing markdown," https://daringfireball.net/2004/03/introducing_markdown, 2004.

[29] S. Dignam, "Are parser generators common? no." https://steve.dignam.xyz/2018/09/18/are-parser-generators-common/, 2018.

[30] P. Eaton, "Parser generators vs. handwritten parsers," https://notes.eatonphil.com/parser-generators-vs-handwritten-parsers-survey-2021.html, 2021.

[31] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/32.988498

[32] A. V. Moura, "Early error detection in syntax-driven parsers," *IBM Journal of Research and Development*, vol. 30, no. 6, pp. 617–626, 1986.

[33] A. V. Aho and J. D. Ullman, "The care and feeding of lr (k) grammars," in *ACM symposium on Theory of computing*, 1971, pp. 159–170.

[34] A. M. Maidl, F. Mascarenhas, and R. Ierusalimschy, "Exception handling for error reporting in parsing expression grammars," in *SBLP*. Springer, 2013, pp. 1–15.

[35] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller, "Parser-directed fuzzing," in *PLDI*. ACM, 2019, p. 548–560.

[36] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard, "Automatic input rectification," in *ICSE*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 80–90.

[37] M. C. Rinard, "Living in the comfort zone," in *OOPSLA*. New York, NY, USA: ACM, 2007, pp. 611–622.

[38] B. Elkarablieh and S. Khurshid, "Juzi: A tool for repairing complex data structures," in *ICSE*. ACM, 2008, pp. 855–858.

[39] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," *SIGPLAN Not.*, vol. 38, no. 11, pp. 78–95, Oct. 2003.

[40] ——, "Data structure repair using goal-directed reasoning," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, May 2005, pp. 176–185.

[41] T. Kuchta, C. Cadar, M. Castro, and M. Costa, "Docovery: Toward generic automatic document recovery," in *ASE*, 9 2014, pp. 563–574.

[42] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault tolerance," *Ieee transactions on computers*, no. 4, pp. 418–425, 1988.

[43] K. Hammond and V. J. Rayward-Smith, "A survey on syntactic error recovery and repair," *Computer Languages*, vol. 9, no. 1, pp. 51–67, 1984.

[44] R. C. Backhouse, *Syntax of programming languages: theory and practice*. Prentice-Hall, Inc., 1979.

[45] S. O. Anderson and R. C. Backhouse, "Locally least-cost error recovery in earley's algorithm," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 3, no. 3, pp. 318–347, 1981.

[46] C. Cerecke, "Locally least-cost error repair in LR parsers," Ph.D. dissertation, 2003.

[47] S. O. Anderson, R. C. Backhouse, E. H. Bugge, and C. Stirling, "An assessment of locally least-cost error recovery," *The Computer Journal*, vol. 26, no. 1, pp. 15–24, 1983.

[48] M. Burke and G. A. Fisher Jr, *A practical method for syntactic error diagnosis and recovery*. ACM, 1982, vol. 17, no. 6.

[49] J. Mauney and C. N. Fischer, "A forward move algorithm for LL and LR parsers," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 79–87, 1982.

[50] S. Zhang, Z. Huang, and E. Wu, "Data cleaning using large language models," *arXiv preprint arXiv:2410.15547*, 2024.

[51] W. Ni, K. Zhang, X. Miao, X. Zhao, Y. Wu, and J. Yin, "Iterclean: An iterative data cleaning framework with large language models," in *ACM Turing Award Celebration Conference - China 2024*, ser. ACM-TURC '24, 2024, p. 100–105.

[52] J. Xu, Y. Fu, S. H. Tan, and P. He, "Aligning the objective of llm-based program repair," in *ICSE*, 2025.