Repairing Property Graphs under PG-Constraints

Christopher Spinrath
Lyon 1 University, Liris CNRS
Lyon, France
christopher.spinrath@liris.cnrs.fr

Angela Bonifati
Lyon 1 University, Liris CNRS & IUF
Lyon, France
angela.bonifati@univ-lyon1.fr

Rachid Echahed
CNRS LIG, Univ. Grenoble Alpes
Grenoble, France
rachid.echahed@imag.fr

ABSTRACT

Recent standardization efforts for graph databases lead to standard query languages like GQL and SQL/PGQ, and constraint languages like Property Graph Constraints (PG-Constraints). In this paper, we embark on the study of repairing property graphs under PG-Constraints. We identify a significant subset of PG-Constraints, encoding denial constraints and including recursion as a key feature, while still permitting automata-based structural analyses of errors. We present a comprehensive repair pipeline for these constraints to repair Property Graphs, involving changes in the graph topology and leading to node, edge and, optionally, label deletions. We investigate three algorithmic strategies for the repair procedure, based on Integer Linear Programming (ILP), a naive, and an LP-guided greedy algorithm. Our experiments on various real-world datasets reveal that repairing with label deletions can achieve a 59% reduction in deletions compared to node/edge deletions. Moreover, the LP-guided greedy algorithm offers a runtime advantage of up to 97% compared to the ILP strategy, while matching the same quality.

PVLDB Reference Format:
Christopher Spinrath, Angela Bonifati, and Rachid Echahed. Repairing Property Graphs under PG-Constraints. PVLDB, 19(6): 1212 - 1225, 2026.
doi:10.14778/3797919.3797929

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://doi.org/10.5281/zenodo.18301604.

1 INTRODUCTION

Repairing errors and inconsistencies is an elemental task when working with large datasets. Applications arise naturally as part of data integration scenarios or if requirements – e.g., the schema – of databases change. Consequently, error repairing has been studied extensively in the literature, for various database models [13].

Property graph languages spanning query languages, i.e. GQL and SQL/PGQ [18, 30], and schema and constraint languages, e.g. PG-Schema and PG-Keys [3, 4], have recently been standardized.

To the best of our knowledge, repairing techniques for property graphs under these schema and constraint languages have not been investigated yet. We embark on the study of *qualitative* error repairing [13] for *property graphs*, that is, error repairing with respect to *constraints*; or more precisely, a subset of PG-Constraints [3, 4], which are part of the PG-Schema formalism and extend PG-Keys.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 6 ISSN 2150-8097.
doi:10.14778/3797919.3797929

To motivate our work, we illustrate property graphs and PG-Constraints by means of a running example.

Example 1.1. Figure 1 depicts the property graph G_e modelling an organisation with persons, tasks, and documents for planning activities. Persons can work on tasks, which reference documents required for the task. Both, nodes and edges, can have multiple labels, and can be equipped with key-value pairs, called *properties*. For instance, the node d_3 has the two labels `document` and `important`, as well as the two properties `#pages` and `access_level`.
Let us consider the following constraint for the property graph G_e.

> "If a person works on a task, which has started and which references directly or indirectly, i.e. recursively, an important document, then the person's access level is at least as high as the (required) access level of the referenced document."

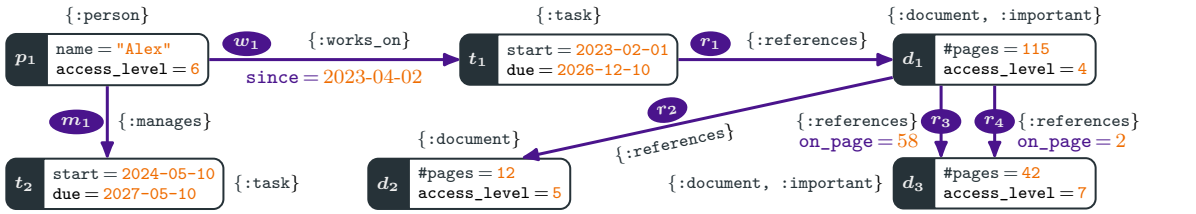
The constraint describes a *graph pattern* in its precondition: The part "a person works on a task" refers to a node labelled `person` and a node labelled `task` which are connected by an edge labelled `works_on`. A *match* of this partial pattern in G_e consists of the nodes p_1, t_1, and the edge w_1. The described pattern also extends to documents which are referenced – directly or indirectly – by the task. The match for the partial pattern above could be extended in several ways: d_1, d_2, and d_3 are all documents referenced by t_1, where d_2 and d_3 are referenced indirectly. In general, reference chains might be of arbitrary length. However, the document d_2 is not labelled `important` and thus it should not be matched.

Lastly, the constraint compares property values, namely the `start` property of t_1 to assert that the task has started, and the `access_level` properties of p_1 and, depending of which node is matched, d_1 or d_3. A possible specification of this constraint as a *PG-Constraint* [4] is illustrated in Figure 1.

G_e violates the above constraint since p_1, w_1, t_1, r_1, d_1, r_4, and d_3 (with d_3 being the important document in question) form a match, the task t_1 has started, but d_3 has a higher access level than p_1.

Contributions. We identify *regular graph pattern calculus (RGPC) constraints* as a subset of PG-Constraints, and introduce an automata model for its patterns to reason about the structure of errors. At the same time, our subset entails major features of PG-Constraints; especially, recursion, complex restrictions of labels for nodes and edges, as well as comparability of node properties. They can express denial constraints, which capture, among others, functional dependencies, including the constraint from Example 1.1. For specifying patterns we use a subset of the graph pattern calculus GPC [26], which captures the core features for patterns in GQL and SQL/PG.

For repairing databases various repair models have been studied in the literature [20], including the addition of new data [5], changing data values [8, 34, 45] – i.e. values of properties – and deletion [12]. The choice of a repair model depends on the setting.



PG-Constraint in GQL

```
FOR x, y MATCH z = (x:person)-[:works_on]->(u:task)(-[:references]->(:document))+(y:document & important) FILTER u.start <= NOW()
MANDATORY x.access_level, y.access_level FILTER x.access_level >= y.access_level
```

Figure 1: A property graph modelling an organisation with persons, tasks, and documents, along with a PG-Constraint

In this paper, we assume the repair model of deletions [see, e.g. 12, 20, 33] in which property graphs are repaired by deleting objects from it, i.e. nodes, edges, or labels. Deletions make sense as a repair model in several applications where inserting new data is undesirable or too risky, for example, due to privacy or security reasons. More concretely, in a social network inserting a friendship could expose private data, but deleting one can be rectified. In a property graph modelling a supply chain, inserting a `sourced_from` relationship between a manufacturer and a product which the manufacturer cannot actually supply can be disastrous for a company, while deleting such a relationship would highlight the issue of unavailability. In a scenario as modelled by the property graph shown in Figure 1, increasing, that is changing, the access level of a person or document is a security risk. Recall that the property graph shown in Figure 1 does not satisfy the constraint from Example 1.1, since the person represented by p_1 works on the task t_1 which references the important document d_3. But the access level of p_1 is not high enough to access d_3. One way to repair the property graph is to delete the edge w_1, effectively removing p_1 from working on t_1. Another possibility is to remove the label `important` from d_3.

In this paper, we propose a comprehensive pipeline (cf., Figure 2) featuring different graph repair options. The repair method follows a *holistic* approach, meaning it detects and repairs all errors simultaneously. This ensures that the fewest possible objects are removed, and their relationships are respected: For example, the deletion of a node entails deleting all its incident edges. However, managing multiple errors – especially those involving long, arbitrary-length paths rather than fixed-length tuples – can be time-consuming. To mitigate this, we propose three algorithms for tackling the underlying combinatorial problem of determining which objects to delete. The first is a naive greedy algorithm, which sequentially repairs each error by deleting an object from it, unless it has already been resolved through prior deletions. The second formulates the problem as an integer linear program (ILP), using a solver to derive an optimal set of objects for deletion. The third one is an LP-guided greedy algorithm which uses non-integer solutions of the LP-relaxation of the ILP to guide the greedy algorithm.

Finally, we gauge the effectiveness of our repair pipeline and analyse the trade-offs associated with the different algorithms and options through experiments conducted on real-world datasets. They reveal that the LP-guided greedy algorithm can outperform the other algorithms, while maintaining the quality, i.e. number of deletions, of the ILP strategy. Permitting label deletions can reduce the number of deletions by up to 59%, but increases the runtime (by up to 5×). Moreover, our experiments reveal that approximate versions of our algorithms can be up to 89% faster, while yielding a good approximation or even a repair.

Structure. In Section 2 we recall some basics on property graphs, and present our subset of the graph pattern calculus (GPC). We then introduce our RGPC constraints in Section 3. In Section 4 we present our pipeline and algorithm(s) for error repairing, and in Section 5 our experiments. Finally, we discuss further related work in Section 6, and conclude in Section 7.

2 PRELIMINARIES

In this section, we provide basics on the data model used in property graphs, and introduce the subset of the pattern calculus GPC [26] which we use as an ingredient for our constraints in Section 3.

2.1 Property Graphs

By $\mathcal{L}, \mathcal{K}, \mathcal{V}$, and $\mathcal{I}$ we denote countably infinite, pairwise disjoint sets of *labels*, *keys* (of properties), *property values*, and *identifiers* (*ids* for short), respectively.

A *property graph* is a tuple $G = (N, E, \rho, \lambda, \pi)$, where

- $N \subset \mathcal{I}$ is a finite set of node identifiers;
- $E \subset \mathcal{I}$ is a finite set of edge identifiers disjoint from N;
- $\rho\colon E \to (N \times N)$ defines the endpoints, i.e. the source and target, for all edges;
- $\lambda\colon (N \cup E) \to 2^{\mathcal{L}}$ is a labelling function that assigns a finite set of labels to node and edge identifiers; and
- $\pi\colon ((N \cup E) \times \mathcal{K}) \rightharpoonup \mathcal{V}$ is a partial function assigning values to pairs of (node or edge) identifiers and keys.

For convenience, we will refer to node and edge identifiers simply as the *nodes* and *edges* of the property graph. Also, we use the term *object* to refer to nodes, edges, and labels on specific nodes or edges.

For example, the nodes of the property graph G_e depicted in Figure 1 are $p_1, t_1, t_2, d_1, d_2, d_3$ and the edges are $w_1, m_1, r_1, r_2, r_3, r_4$. The endpoints of the edge w_1 are p_1 (the source) and t_1 (the target). We note that it is possible that a node or an edge does not have any labels. In that case, λ assigns the empty set (of labels).

A *path* in a property graph $G = (N, E, \rho, \lambda, \pi)$ is a non-empty alternating sequence $v_0 e_1 v_1 \dots e_n v_n$ where $v_0, \dots, v_n \in N$ are nodes and $e_1, \dots, e_n \in E$ are edges such that $\rho(e_i) = (v_{i-1}, v_i)$ holds for all

$i \in \{1, \ldots, n\}$. For example, the sequence $p_1 w_1 t_1 r_1 d_1 r_3 d_3$ is a path in G_e. We emphasize that every path starts and ends with a node. The *length* of a path $v_0 e_1 v_1 \ldots e_n v_n$ is the number of edges n. An empty path consists of exactly one node and has length 0. Two paths $v_0 e_1 \ldots e_n v_n$ and $v'_0 e'_1 \ldots e'_m v'_m$ can be *concatenated* if $v_n = v'_0$ holds. The resulting path is then $v_0 e_1 \ldots e_n v_n e'_1 \ldots e'_m v'_m$. The *trace* $\lambda(P)$ of a path $P = v_0 e_1 v_1 \ldots e_n v_n$ is $\lambda(P) = \lambda(v_0)\lambda(e_1)\lambda(v_1)\ldots\lambda(e_n)\lambda(v_n)$. For example, the trace of the path $p_1 w_1 t_1 r_1 d_1 r_3 d_3$ is

$$\{\texttt{person}\}\{\texttt{works_on}\}\{\texttt{task}\}\{\texttt{references}\}$$
$$\{\texttt{document}, \texttt{important}\}\{\texttt{references}\}\{\texttt{document}, \texttt{important}\}.$$

2.2 Regular Graph Pattern Calculus

In the following we introduce the syntax and semantics of a proper subset of GPC, a graph pattern calculus, which captures the core features of the pattern sublanguage shared by GQL and SQL/PGQ [26]. We present our subset, which we call *RGPC* (short for *regular GPC*), by means of examples.

Example 2.1. The following RGPC pattern matches all pairs x, y of nodes such that x has the label `document` and there is a path z from x to y on which every edge has the label `references`.

$$z = (x : \texttt{document})\ [\xrightarrow{:\texttt{references}}]^*\ (y)$$

We explain the components of the pattern in Example 2.1, as well as some variations, in a bottom-up fashion. The basic building blocks of RGPC patterns are *node* and *edge patterns*. A node pattern has the form $(x : \varphi)$ where x is a *node variable* and φ is a *label expression* which restricts the labels of the nodes matching the pattern. Both x and φ are optional. The node patterns in Example 2.1 are $(x : \texttt{document})$, and (y). In the latter pattern the labels of y are not restricted, the node may have any label(s). An example for a node pattern without variable is $(:\texttt{a})$. If x and φ are both omitted, we just write $()$. In general, φ can be any propositional formula connecting labels from the set $\mathcal{L}$ with the operators $\wedge$ (conjunction), $\vee$ (disjunction), and $\neg$ (negation). For example, $(:(\texttt{a} \wedge \texttt{b}) \vee \neg\texttt{c})$ matches every node which has labels `a` and `b` or not label `c`.

Similarly to node patterns, edge patterns have the form $\xrightarrow{:\varphi}$, although we do not consider variables in edge patterns. In Example 2.1, there is one edge pattern, namely $\xrightarrow{:\texttt{references}}$.

Path Patterns. Node and edge patterns can be combined into *path patterns* using *concatenation*, *union*, and *repetition*.

For example, a *concatenation* of two node and an edge pattern is

$$(x : \texttt{document}) \xrightarrow{:\texttt{references}} (y)$$

which matches all pairs x, y of nodes such that x has the label `document` and there is an edge labelled `references` from x to y.

The *union* of two patterns can be built if they *do not contain node variables*. For instance, $\xrightarrow{:\texttt{a}} \cup \xrightarrow{:\texttt{c}}\xrightarrow{:\texttt{c}}$ states that there is an edge labelled `a`, *or* a path of length two on which all edges are labelled `c`.

For *repetitions* we consider the classic Kleene-star operator. For example, $\left[\xrightarrow{:\texttt{a}}\right]^*$ specifies that the path specified by $\xrightarrow{:\texttt{a}}$ can be repeated arbitrarily often (including 0 times). We also write $\left[\xrightarrow{:\texttt{a}}\right]^+$ as shorthand for $\xrightarrow{:\texttt{a}}\left[\xrightarrow{:\texttt{a}}\right]^*$. Similar to union, repetition is limited to patterns that do not contain any node variables. Additionally, patterns below a repetition must not match a path of length 0.

Of course, concatenation, union, and repetition can be nested. To improve readability (round brackets are used in node patterns) we use square brackets for disambiguation.

Graph Patterns. Finally, *graph patterns* are conjunctions of path patterns, associated with pairwise distinct *path variables*.[1] For example, the following graph pattern consists of two path patterns

$$z_a = (x : \texttt{b}) \xrightarrow{:\texttt{a}} (y),\ z_c = (y) \xrightarrow{:\texttt{c}}{}^+ (x)$$

with variables z_a and z_c. It specifies that there are nodes x, y in the property graph such that x is labelled `b`, there is an edge (x, y) labelled a, and a path from y to x on which every edge is labelled c. Let us point out that path and node variables are disjoint.

Example 2.2. Consider the following graph pattern where `p`, `w`, `t`, `d`, `r` and `i` are shorthands for the labels `person`, `works_on`, `task`, `document`, `references` and `important`, respectively.

$$z = (x : \texttt{p}) \xrightarrow{:\texttt{w}} (:\texttt{t})\left[\xrightarrow{:\texttt{r}} (:\texttt{d})\right]^+ (y : \texttt{d} \wedge \texttt{i})$$

It specifies the topology from Example 1.1: person x works on a task which (possibly indirectly) references an important document y. Note that all nodes on the subpath from the task to y must be labelled `document`, except for the first node, representing the task.

Semantics. A *match* for a (graph) pattern q in a property graph G is a mapping μ that maps the node and path variables of q to nodes and paths in G, respectively, which constitutes an answer to q on G. We refer to the literature [26] for a formal definition.

Example 2.3. A match for the pattern in Example 2.2 in the property graph depicted in Figure 1 is the mapping μ with $\mu(x) = p_1$, $\mu(y) = d_3$, and $\mu(z) = p_1 w_1 t_1 r_1 d_1 r_3 d_3$.

3 CONSTRAINTS FOR PROPERTY GRAPHS

With the preliminaries in place, we are now ready to introduce the constraints we study in this paper. We first explain the idea by means of an example, and then make the terms more precise.

Example 3.1. We consider the following constraint as a running example in this section: "Every document has a greater access level than any other document referenced (directly or indirectly) by it, that has not been edited since 2020."

The core ingredient of a constraint is an RGPC graph pattern, incorporating characteristics of established graph query languages such as GQL and SQL/PGQ, notably recursion. This allows us to express the topology described in Example 3.1: The constraint affects all pairs of documents, say x and y, such that x references (directly or indirectly) y. The following RGPC pattern models this topology.

$$z = (x : \texttt{document})\ [\xrightarrow{:\texttt{references}}]^+\ (y : \texttt{document})$$

The RGPC pattern is complemented with two more ingredients to yield a constraint. Both ingredients can refer to node variables.

The first can be understood as an additional filter on the matches of the pattern. For instance, the predicate $x \neq y$ ensures that x and y in the above pattern are matched to different documents. Furthermore, the constraint in Example 3.1 only affects documents y which have not been edited since 2020. Assuming nodes labelled `document`

[1] In the original definition of GPC, path variables are optional, but here they are obligatory. They also do not have to be distinct. We demand them to be distinct to avoid "hidden" path equality constraints.

have a property for the last time it was edited, i.e. a timestamp, this can be ensured by the predicate $y.\texttt{edited} \leq \texttt{2020-12-31}$. In general, it is possible to supply any number of predicates (or none), in which case all of them have to hold for a match to pass the filter.

The last ingredient also consists of predicates and states which conditions have to hold, for every match of the pattern that passes the filter. For our example, the condition is the set consisting of the predicate $x.\texttt{access_level} \geq y.\texttt{access_level}$.

Overall the constraint described in Example 3.1 can be written as follows using the shorthands from Example 2.2.

$$z = (x:\mathsf{d})\,[\xrightarrow{:\mathsf{r}}]^+\,(y:\mathsf{d}); \{x \neq y,\ y.\texttt{edited} \leq \texttt{2020-12-31}\} \Rightarrow \{x.\texttt{access_level} \geq y.\texttt{access_level}\}$$

In general, the constraints we study in this paper have the shape $q; F \Rightarrow C$ where q is an RGPC graph pattern, and F and C are sets of predicates modelling the filter and conditions of the constraint, respectively. More precisely, we consider all comparison predicates of GQL [30, Section 19.3]: $x_i = y_j$, $x_i \neq y_j$, $x_i.\texttt{propA} \oplus y_j.\texttt{propB}$, and $x_i.\texttt{propA} \oplus c$ where x_i and y_j are node variables that occur in q, $\oplus \in \{=, \neq, \leq, \geq, <, >\}$, and c is a value from $\mathcal{V}$.

The semantics of these predicates are as usual, with the noteworthy trait that all mentioned properties have to exist. For example, the filter and condition in the constraint above imply the existence of the `edited` and `access_level` property, respectively. A formal definition is available in the extended version [42, Appendix A.2]. A match μ satisfies a set of predicates if it satisfies all predicates in the set. A property graph G satisfies a constraint $q; F \Rightarrow C$ if every match μ in G for q satisfies C or not F.

Example 3.2. The following constraint σ is described informally in Example 1.1.

$$\underbrace{z = (x:\mathsf{p}) \xrightarrow{:\mathsf{w}} (u:\mathsf{t})\big[\xrightarrow{:\mathsf{r}} (:\mathsf{d})\big]^+ (y:\mathsf{d} \wedge \mathsf{i});}_{q}$$
$$\{u.\texttt{start} \leq \texttt{now}\} \Rightarrow \{x.\texttt{access_level} \geq y.\texttt{access_level}\}$$

Here q is the RGPC pattern explained in Example 2.2, except for the additional node variable u. We use "now" as a place holder for the current timestamp. The property graph illustrated in Figure 1 does *not* satisfy the constraint, because the match in Example 2.3 satisfies the filter (the start date of the task is in the past), but not $\{x.\texttt{access_level} \geq y.\texttt{access_level}\}$. Indeed, x is matched to p_1 and y to d_3 and we have $p_1.\texttt{access_level} = 6 \not\geq 7 = d_3.\texttt{access_level}$.

As discussed in the introduction, the constraint σ from Example 3.2 can be expressed as the PG-Constraint in Figure 1: The RGPC pattern is translated into a GQL, ASCII-art-like pattern, and the filter and consequence are expressed as `FILTER` clauses. Analogously, all RGPC constraints can be written as PG-Constraints.

Example 3.3. So far our examples were implication dependencies, a subclass of denial constraints. Using an unsatisfiable predicate like $x \neq x$, which we simply denote as `false`, we can also express denial constraints which do not fall into this subclass. For instance, the following denial constraint states that there are no cyclic references between documents.

$$(x:\mathsf{d})\big[\xrightarrow{:\mathsf{r}} (:\mathsf{d})\big]^+(x); \emptyset \Rightarrow \{\texttt{false}\}$$

4 REPAIRING PROPERTY GRAPHS

This section is dedicated to the presentation of our repair pipeline, which is depicted in Figure 2. We start by introducing the notion of a *repair of a property graph* in Section 4.1. In Section 4.2 we present the *base pipeline*, that is, our pipeline without any of the optional steps it features. They are introduced in Sections 4.3 and 4.4.

4.1 Property Graph Repairs

As discussed in the introduction, motivated by applications where changing or adding objects is too risky due to, e.g. privacy or security reasons, we repair property graphs by deleting objects – i.e. nodes, edges, or labels – from it.

Example 4.1. Consider the property graph G_e depicted in Figure 1 and the constraint σ from Example 3.2. The property graph G_e does *not* satisfy the constraint, because "Alex" works on task t_1 which indirectly references, via document d_1, the important document d_3. However, "Alex" has a strictly smaller access level than d_3.

To repair G_e, under the repair model of deletions, it suffices to delete one of the nodes p_1, t_1, d_1, or d_3, or one of the edges w_1 or r_1 between them. Note that deleting either r_3 or r_4 is *not* sufficient (but deleting both is). Moreover, all of these options will change the *topology* of G_e.

Another option is to delete a label of a node or edge, for instance, here it would suffice to remove the label `:important` from d_3. This option does not change the topology of the graph at all.

To properly specify repairs, we will employ *subgraphs*. Intuitively, a subgraph is a property graph obtained by deleting nodes, edges, labels (or properties) from a given property graph.

Definition 4.2 (Subgraph). Let $G = (N, E, \rho, \lambda, \pi)$ be a property graph. A property graph $G' = (N', E', \rho', \lambda', \pi')$ is called a *subgraph* of G if the following five conditions hold : (i) $N' \subseteq N$, (ii) $E' \subseteq E$, (iii) $\rho'(o') = \rho(o') \cap (N' \times N')$ for all $o' \in E'$, (iv) $\lambda'(o') \subseteq \lambda(o')$ for all $o' \in N' \cup E'$ and (v) If $\pi'(o', k)$ is defined for a property key k and $o' \in N' \cup E'$, then so is $\pi(o', k)$ and $\pi'(o', k) = \pi(o', k)$. That is, all nodes, edges, labels, and properties (as well as their values) of G' are also present in G. If all nodes and edges in G' have exactly the same labels and property key/value pairs as in G, i.e. if $\lambda'(o') = \lambda(o')$ and $\pi'(o', k) = \pi(o', k)$, for all $o' \in N' \cup E'$ and property keys k, then we call G' a *topological subgraph* of G. We have that G' is a *proper subgraph* of G, if $G \neq G'$ holds.

For example, removing the label `:important` from node d_3 in the property graph G_e depicted in Figure 1 results in a proper subgraph, say $G_\mathsf{e}^\star$. The subgraph $G_\mathsf{e}^\star$ is *not* topological because the node d_3 is also present in the original graph G_e but the labels of d_3 in $G_\mathsf{e}^\star$ are not the same as in G_e. Another example is the topological subgraph obtained from G_e by deleting the edges r_3 and r_4.

For being a *repair*, a subgraph has to satisfy all constraints, and, in the spirit of previous work [see, e.g. 12, Definition 2.2], it should be maximal, i.e. there should be no unnecessary deletions.

Definition 4.3 (Repair). Given a property graph G and a set Σ of constraints, a *repair* of G is a subgraph G' of G that

a) satisfies Σ; and
b) is *maximal*, that is, there is *no* subgraph G'' of G such that G' is a proper subgraph of G'' and G'' satisfies Σ.

When a subgraph G_a of G satisfies Condition a) but not necessarily Condition b), we call G_a an *approximate* repair. A *topological (approximate) repair* is defined analogously to a(n approximate) repair, except that all subgraphs involved are topological subgraphs.

Example 4.4 (Continuation of Example 4.1). Let G'_e be the subgraph of the property graph G_e obtained by deleting the edge r_1 between t_1 and d_1. We observe that G'_e is a topological repair. But it is *not* a repair, because the subgraph G''_e obtained from G_e by removing (just) the `:references` label from r_1 is also a repair, and G'_e is a subgraph of G''_e. Thus, Condition b) of Definition 4.3 is violated. It is, however, an approximate repair, since it satisfies Condition a).

Similarly, removing the node d_3 is not a repair, since it suffices to remove either the `:document` or the `:important` label. Observe that removing a node always implies removing all edges involving this node as well: otherwise, the result is not a well-defined property graph. Thus, the removal of d_3 leads to the removal of the edges r_3 and r_4. But then the resulting subgraph is also *not* a topological repair: it is possible to add back d_3, as long as the edges are not added back. We discuss this effect in more detail later on.

4.2 The Base Pipeline

We are now ready to present our repair pipeline, which is illustrated in Figure 2. It takes a set Σ of RGPC constraints and a property graph G as input, and has 6 steps, the last of which carries out the deletion operations. Two of these steps are optional, namely steps 2 and 3: they lead to label removals and a controllable trade-off between quality and runtime, respectively. In the following, we first discuss the *base pipeline*, consisting of steps 1, 4, 5, and 6.

Error Retrieval (Step 1). Intuitively, we understand an error as a set of objects witnessing that a constraint is not satisfied by the property graph. Since all path patterns of an RGPC constraint are associated with a path variable, an error can be obtained from the range of a match which satisfies the filter but not the consequence of the constraint.

Consider again the constraint from Example 3.2 and the property graph depicted in Figure 1. The property graph does not satisfy the constraint, because there are two matches of the constraint's pattern which pass the filter and do not satisfy the condition of the constraint: The first match corresponds to the path $p_1w_1t_1r_1d_1r_3d_3$ (cf. Example 2.3), and the second one to the path $p_1w_1t_1r_1d_1r_4d_3$. As illustrated in Figure 3 (1), (2), the sets of nodes and edges occurring on these paths, namely $O_1 = \{p_1, w_1, t_1, r_1, d_1, r_3, d_3\}$ and $O_2 = \{p_1, w_1, t_1, r_1, d_1, r_4, d_3\}$ constitute *topological errors*.

Definition 4.5 (Topological Error). Let $G = (N, E, \rho, \lambda, \pi)$ be a property graph, $q; F \Rightarrow C$ be a constraint with path variables $z_1, \ldots, z_k$, and μ be a match for q in G that witnesses G *not* satisfying the constraint. The set O_μ, consisting of all nodes and edges, $o \in N \cup E$, for which there is an i, $1 \leq i \leq k$, such that o occurs in $\mu(z_i)$, is called a *topological error* of the constraint in G.

Error detection by means of graph queries has already been introduced in previous work [4]. We put this approach into practice and delegate error detection and retrieval to a graph database system by rewriting every constraint into an *error query*, which asks for all paths involved in an error. For example, the constraint from Example 3.2 is rewritten into the following GQL error query.

```
MATCH z = (x:person)-[:works_on]->(u:task)
          (-[:references]->(:document))+
          (y:document & important)
FILTER u.start IS NOT NULL AND u.start <= NOW()
AND (x.access_level < y.access_level
     OR x.access_level IS NULL OR y.access_level IS NULL)
RETURN z
```

Hypergraph Construction (Step 4). For repairing the property graph, the idea is to pick one object from each error and delete it. Since every error corresponds to a match, deleting a single object from it suffices to eliminate the match; hence, the resulting property graph satisfies the constraint(s). However, the outcome is not necessarily a repair. For instance, selecting r_1 and r_4 from the errors O_1 and O_2 discussed above, respectively, does not yield a repair, because it is possible to add r_4 back without violating the constraint. This is because r_1 occurs in both errors. We already discussed in Example 4.4 that removing nodes can have a similar effect.

To address this issue it helps to understand a collection of (topological) errors as a hypergraph, called *conflict hypergraph*. A *hypergraph* is a pair $\mathcal{H} = (\mathcal{W}, \mathcal{E})$ consisting of a set $\mathcal{W}$ of vertices[2] and a set $\mathcal{E} \subseteq 2^{\mathcal{W}}$ of hyperedges. The conflict hypergraph for our running example is depicted in Figure 3 (3).

Definition 4.6 (Topological Conflict Hypergraph). Given a property graph $G = (N, E, \rho, \lambda, \pi)$, a set Σ of constraints, the *topological conflict hypergraph* of G w.r.t. Σ is the hypergraph $\mathcal{H} = (\mathcal{W}, \mathcal{E})$ whose edges are precisely all topological errors of all constraints in Σ in G, and where $\mathcal{W} = \bigcup_{O \in \mathcal{E}} O$ is the union of all these edges.

Computing a Repair (Step 5c). To repair a property graph, the idea is to select vertices from the conflict hypergraph with high-inbetweenness, i.e. vertices which participate in a high amount of errors. At the same time we want to select a minimal number of vertices. This amounts to computing a minimal vertex cover:

A *vertex cover* of a hypergraph $\mathcal{H} = (\mathcal{W}, \mathcal{E})$ is a set $V \subseteq \mathcal{W}$ such that $V \cap O \neq \emptyset$, for each $O \in \mathcal{E}$. A vertex cover V is minimal if there is no vertex cover V' with $V' \subsetneq V$.

Intuitively, a minimal vertex cover tells us which objects to delete to obtain a repair. For example, a vertex cover for the conflict hypergraph with hyperedges O_1 and O_2 from above is $V = \{r_1\}$. Trivially, no subset of V is a vertex cover, and removing r_1 yields a repair, as discussed above. The set $V \cup \{r_4\} = \{r_1, r_4\}$ is not a minimal vertex cover, since it contains V. And indeed, removing r_1 and r_4 does not yield a repair for the same reason: r_4 can be added (back).

However, this does not address the issue of node removal implying removal of incident edges. For instance, $\{d_3\}$ is a minimal vertex cover, but as discussed in Example 4.4, removing d_3 implies the removal of r_3 and r_4, and does not yield a repair. Therefore, the deletion of nodes should be avoided, unless it is necessary, e.g. because an error consists only of nodes. We realize this by assigning a weight to each vertex of the conflict hypergraph $\mathcal{H}$. The weight $w(n)$ of a node n of the property graph is the number of edges incident to n plus one. The weight $w(e)$ of an edge e is simply 1. We are then interested in a *minimum weight vertex cover* [see e.g. 46] of $\mathcal{H}$. The weight of a vertex cover V is $w(V) = \sum_{v \in V} w(v)$. A *minimum weight vertex cover* V is a vertex cover whose weight $w(V)$

[2]We call them vertices to differentiate them from the nodes of property graphs.

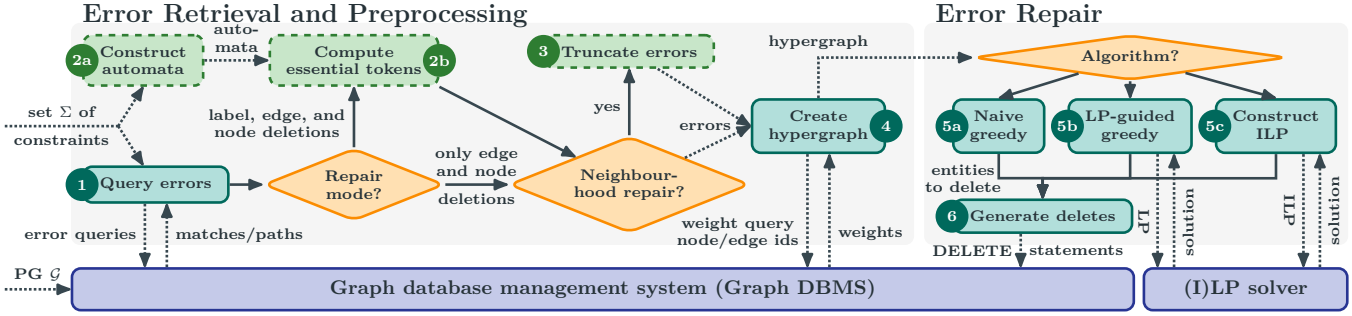


Figure 2: Our repair pipeline for property graphs consists of 6 steps, where steps 2 and 3 are optional. 2a and 2b are either both enabled or not, while 5a, 5b and 5c are alternatives. Solid edges indicate control flow, while dotted edges indicate communication.

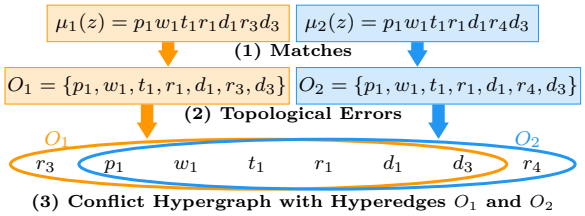


Figure 3: Different encodings of topological errors

is the minimum among all vertex covers (of $\mathcal{H}$). Note that, if all vertices have the same weight, minimum weight vertex covers coincide with minimum vertex covers[3], which is a stronger notion than that of a minimal vertex cover. Each minimum vertex cover is in turn a minimal vertex cover (but not vice versa). We have the following result, whose proof is available in the extended version [42, Appendix B.2].

Proposition 4.7. *Given a minimum weight vertex cover V, removing[4] all objects in V from G yields a topological repair of G.*

The following integer linear program (ILP) with integer variables x_v, for each vertex $v \in \mathcal{W}$, encodes the problem of finding a minimum weight vertex cover.

$$\text{minimize} \sum_{v \in \mathcal{W}} w(v) \cdot x_v \quad \text{subject to} \sum_{v \in O} x_v \geq 1 \ \text{ for all } O \in \mathcal{E}$$
$$\text{and } x_v \in \{0, 1\} \ \text{ for all } v \in \mathcal{W}$$

Given a solution for this ILP, that is, a function that assigns all variables x_v into $\{0, 1\}$, corresponds to a minimum weight vertex cover V: It consists of all vertices v for which x_v is mapped to 1.

Overall, a repair can be computed by rewriting every constraint into an error query, querying for errors, and then solving the ILP above. Since there are at most exponentially many hyperedges and integer linear programming is in NP, the complete procedure runs in non-deterministic exponential time (data complexity).

Greedy Repairs (Steps 5a and 5b). Despite ILP solvers being reasonably fast in practice, it can be desirable to trade the quality of a repair – in our case, the number of objects deleted – for a better runtime. A common approach to achieve this are so called greedy algorithms, of which we consider two.

The first we call *naive greedy algorithm*. It computes a vertex cover V_g with a small weight in two phases: the *selection* and the *trimming* phase. In the selection phase the algorithm iterates over all hyperedges to select candidates for the vertex cover: If the current hyperedge contains a vertex, which has already been selected and has minimal weight among the vertices in the hyperedge, the hyperedge is skipped. Otherwise, the algorithm selects an arbitrary vertex from the hyperedge with minimal weight among the vertices in the hyperedge. The selected vertices form a vertex cover V_g, because every hyperedge contains at least one selected vertex.

However, the algorithm may end up selecting too many vertices, and thus V_g might not be *minimal*. Thus, this phase yields an approximate repair. To obtain a repair, the algorithm attempts, in the trimming phase, to reduce the number of selected vertices again. It does this by checking, for every selected vertex v, ordered by weight in descending order, whether there is a hyperedge E for which v is the only selected vertex, i.e. for which $V_g \cap E = \{v\}$ holds. If there is *no* such hyperedge E, then v has to be removed from V_g.

Consider again the two errors $O_1 = \{p_1, w_1, t_1, r_1, d_1, r_3, d_3\}$ and $O_2 = \{p_1, w_1, t_1, r_1, d_1, r_4, d_3\}$. The naive greedy algorithm might compute $V_g = \{r_3, r_4\}$ in the first phase. In the second phase, V_g is not changed, because r_3 and r_4 do not occur in O_2 and O_1, respectively. Thus, the vertex cover V_g is minimal because neither r_3 nor r_4 can be removed. However, it is not a minimum weight vertex cover, since, e.g., the minimum vertex cover $\{r_1\}$ has a smaller weight.[5] While the naive greedy algorithm does not necessarily compute a minimum weight vertex cover, picking vertices with minimal weight is sufficient to compute a (topological) repair. Since the algorithm iterates over all errors, it runs in exponential time.

Proposition 4.8. *Given a minimal vertex cover V_g computed by the naive greedy algorithm, removing all objects in V_g from G yields a topological repair of G.*

A proof is available in the extended version [42, Appendix B.2].

As for the second greedy algorithm, we consider an *LP-guided greedy algorithm* which combines a solution for the *LP-relaxation* of the ILP with the naive greedy algorithm. Here the LP-relaxation is the linear program (LP) obtained from the ILP by replacing the

[3] V is a minimum vertex cover, if there is no vertex cover V' with $|V'| < |V|$.
[4] Recall that removing a node implies removing all its incident edges in G.
[5] It is also possible that the naive greedy algorithm picks r_1 instead of r_3 and r_4 in which case it would yield the minimum weight vertex cover $\{r_1\}$.

conditions $x_v \in \{0, 1\}$ with inequalities $0 \leq x_v \leq 1$ and allowing the x_v to assume real values. It is well-known that LPs can be solved in polynomial time, unlike ILPs. The algorithm then selects any vertex v for which $x_v > 0$ holds as a candidate, and then proceeds as the naive greedy algorithm.

4.3 Optional Step 2: Label Deletions

We continue with the description of optional Step 2 in Figure 2 of our repair pipeline, which allows us to obtain non-topological repairs by deleting labels (in addition to nodes and edges).

Our repair pipeline – whether the ILP or a greedy algorithm is used – deletes nodes only if errors contain isolated nodes (aka paths of length 0). Otherwise, an edge is deleted, since edges have minimal weight 1. This can lead to the presence of (potentially many) isolated nodes in a repair, when all incident edges of a node are deleted. To avoid this case, we study repairs which are not necessarily topological. More precisely, we extend our approach to include the removal of labels to obtain a repair, as discussed in Example 4.1. Instead of deleting (many) incident edges of a node, it can suffice to remove a single label of a node. Note that it is not always possible to obtain a repair by only removing labels, i.e., if a constraint does not refer to any label. Thus, it may still be necessary to delete nodes and/or edges, in addition to deleting labels.

The idea is to extend topological errors O_μ, which initially contain the nodes and edges of the property graph occurring in the co-domain of a match μ, by pairs (o, ℓ) of objects o, i.e. a node or an edge, and labels ℓ. We call these pairs (o, ℓ) *tokens*. The intended meaning is that the error can be repaired by removing the label ℓ from node (or edge) o. The edges of the (not necessarily topological) conflict hypergraph are then all extended sets – which we will simply call *errors* in the following. We note that, in general, there are multiple extensions of a single topological error. In particular, there are more errors than topological errors.

In the following we discuss how to compute errors given a match. We start with the simple case of a single constraint with a single path pattern. Consider once again the constraint from Example 3.2. Recall that it consists of one path pattern with path variable z. We can observe that removing the `person` label from the node p_1, or removing the `works_on` from the edge w_1 yields a repair. Indeed, every path matching the pattern of the constraint has to start with a node labelled `person` followed by an edge labelled `works_on`, and p_1 and w_1 are the only objects with these labels. Thus, $(p_1, \texttt{person})$ is a token we are looking for. The same is true for $(w_1, \texttt{works_on})$. We say that these tokens are *essential*.

Automata Construction (Step 2a). To compute essential tokens automatically for arbitrary RGPC path patterns, we construct automata, which we call *RGPC automata* and introduce next, by means of an example. Consider the path pattern

$$(x\,\texttt{:p}) \xrightarrow{\texttt{:w}} (u\,\texttt{:t}) \Big[\xrightarrow{\texttt{:r}} (\texttt{:d}) \Big]^{+} (y\,\texttt{:d} \wedge \texttt{i})$$

from Example 3.2. The RGPC automaton for this path pattern is illustrated in Figure 4. Note that the transitions are labelled with label expressions, i.e. propositional formula connecting labels – instead of just (single) labels. For instance, the transition from state 5 to 6 is labelled with `d` $\wedge$ `i`, which is the label expression of the node pattern $(y\,\texttt{:d} \wedge \texttt{i})$. Thus, RGPC automata belong to the family of so called *symbolic* automata [see, e.g., 14]. A transition can be taken if the label expression is satisfied by the *set* of labels of a node or edge. For instance, the transition with the formula `d` $\wedge$ `i` actually represents several transitions: one for each set of labels that contains `d` and `i`. Notably, the transition from State 5 to State 4 can also be taken if `d` and `i` are both present. This allows the automaton to read traces of paths, e.g. the trace {p}{w}{t}{r}{d, i}{r}{d, i} of the path $p_1 w_1 t_1 r_1 d_1 r_3 d_3$ from the end of Section 2.1.

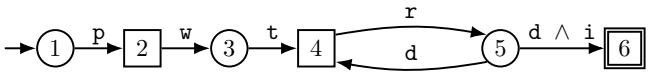


Figure 4: Automaton for the RGPC pattern from Example 3.2

Furthermore, unlike classical automata, the RGPC automaton illustrated in Figure 4 has two kinds of states: Those drawn as a circle and those drawn as a square, indicating that the next transition reads the label(s) of a node or the label(s) of an edge, respectively. For instance, the outgoing transition of State 1 originates from the node pattern $(\texttt{:p})$, and the outgoing transition of State 2 from the edge pattern $\xrightarrow{\texttt{:w}}$. In particular, the RGPC automaton reads node and edge labels in alternation, and can thus read the trace of a path.

Finally, note that variables of the path pattern do not play a role for the automaton construction: We use RGPC automata to rescan paths returned during the error retrieval step (Step 1). We thus already know that such a path is a match, and we are only interested in identifying labels for deletions.

The formal RGPC automata construction is available in the extended version [42, Appendix B.1].

Algorithm 1: Computing Errors with Essential Tokens for a Single Path

Input: Match μ, path pattern $z = \mathcal{P}$, $\mu(z) = v_0 e_1 v_1 \dots e_n v_n$
Output: Set of errors $\mathcal{E}_\mu$ with essential tokens

$\mathcal{A} \leftarrow$ automaton for $\mathcal{P}$;
$O_\mu \leftarrow \{v_0, e_1, v_1, \dots, e_n, v_n\}$; // topological error
$\mathcal{E}_\mu \leftarrow \emptyset$; // set of errors
foreach *accepting run r of $\mathcal{A}$ on path $\mu(z)$* **do**
  $O_r \leftarrow \emptyset$;
  foreach *$o \in O_\mu$ and label ℓ* **do**
    if *removing ℓ from o invalidates r* **then**
      $O_r \leftarrow O_r \cup \{(o, r)\}$; // (o, ℓ) is essential
  $O_{\mu,r} \leftarrow O_\mu \cup O_r$; $\mathcal{E}_\mu \leftarrow \mathcal{E}_\mu \cup \{O_{\mu,r}\}$;
return $\mathcal{E}_\mu$

Computing Essential Tokens (Step 2b). Our algorithm for computing errors for a single path pattern $\mathcal{P}$ is outlined in Algorithm 1. It uses the automata constructed in Step 2a to identify essential tokens, as follows: Let z be the path variable of $\mathcal{P}$ and $\mathcal{A}$ be the automaton constructed for it, and μ be a match of $\mathcal{P}$. Then $\mathcal{A}$ accepts the trace of the path $\mu(z)$. To obtain essential tokens, we consider every accepting run r of $\mathcal{A}$ on the trace of $\mu(z)$: if removing a label ℓ from an object o on the path $\mu(z)$ invalidates the run, it is *essential for this run*. For example, the trace {p}{w}{t}{r}{d, i}{r}{d, i} is accepted by the automaton depicted in Figure 4. Removing the label `i` from the last set in the trace invalidates all accepting runs, because

the automaton can no longer take the transition from state 5 to 6 after reading the second {r} (and if the transition is taken earlier the automaton cannot read the whole trace).

For invalidating the match μ, all accepting runs have to be invalidated. Thus, for each accepting run r, we extend O_μ to the error $O_{\mu,r}$ by adding all tokens which are essential for r. Note that, if the pattern does not contain any labels, there are no essential tokens for any run. In that case, the original error O_μ is kept as is.

Suppose now the constraint consists of a graph pattern q with k path patterns, with path variables $z_1, \ldots, z_k$. Let μ be a match witnessing that the constraint is not satisfied. Let further $\mathcal{A}_i$ be the automaton for pattern $i \in \{1, \ldots, k\}$, and $r_1, \ldots, r_k$ be accepting runs of $\mathcal{A}_1, \ldots, \mathcal{A}_k$ on the traces of the paths $\mu(z_1), \ldots, \mu(z_k)$. While it suffices to invalidate one of these runs, to invalidate *the combination* $r_1, \ldots, r_k$, we observe that there might be other combinations involving the same runs: Say we invalidated an accepting run r_j by deleting some label. Then the runs $r_1, \ldots, r_j, \ldots, r_k$ do no longer witness that μ is a match. However, there might be another run r'_j such that $r_1, \ldots, r'_j, \ldots, r_k$ still witness that μ is a match.

We thus create an error for every combination $r_1, \ldots, r_k$ of accepting runs. More precisely, we condense the sets $O_{\mu,r_1}, \ldots, O_{\mu,r_k}$ into a single error $O_{\mu,r_1,\ldots,r_k} = O_{\mu,r_1} \cup \cdots \cup O_{\mu,r_k}$. The sets $O_{\mu,r_1,\ldots,r_k}$, for each match μ and all combinations of runs $r_1, \ldots, r_k$, are then the hyperedges of the *conflict hypergraph*. This way all combinations have to be invalidated by removing (at least) one object.

Since removing a node or edge implies the removal of all its labels, we adapt the weights accordingly: For an edge, the weight is the number of its labels plus 1, and for a node the sum of the number of its labels, the weights of its incident edges, and 1 (for itself). A token (o, ℓ) represents a single label, and thus has weight 1.

We emphasize that Proposition 4.7 does *not* carry over, since the formulas occurring in an RGPC pattern can contain negations: Thus, deleting labels can lead to new matches, and hence, to new errors. We call an RGPC constraint *positive* if *no* label expression in its pattern contains any negation. We then have the following.

Proposition 4.9. *A repair with label removals can be computed in non-deterministic exponential time (data complexity), if the given RGPC constraints are positive.*

4.4 Optional Step 3: Neighbourhood Errors

The optional Step 3 attempts to reduce the runtime by limiting the size of errors, and thus also the number of errors. As a trade-off the repair pipeline yields, in general, an approximate repair.

The size of errors does depend on the size of the considered property graph, and the number of errors can be exponential. We address this problem by introducing neighbourhood errors: In a nutshell, the idea is to repair a property graph by removing objects "close" to the endpoints of the paths inducing errors.

For an integer $k \geq 1$, which can intuitively be understood as a radius around the endpoints of a path, the *k-endpoint neighbourhood* of a path $v_0 e_1 v_1 e_2 \ldots e_n v_n$ consists of the objects which have distance at most k from one of the endpoints v_0 and v_n, that is, the objects $v_0, e_1, v_1, \ldots, e_k, v_k$ and $v_{n-k}, e_{n+1-k}, v_{n+1-k}, \ldots e_n, v_n$.

Given an RGPC constraint $q; F \Rightarrow C$ with ℓ path variables $z_1, \ldots, z_\ell$, and a match μ for q witnessing that the constraint is *not* satisfied, the *topological k-neighbourhood error* O^k_μ consists of all nodes and edges o, for which there is a path variable z_i, $1 \leq i \leq \ell$, such that o occurs in the k-endpoint neighbourhood of $\mu(z_i)$.

When Step 3 is enabled, the pipeline uses (topological) k-neighbourhood errors in place of (topological) errors. For example, consider once more the constraint from Example 3.2, the property graph depicted in Figure 1, and recall that there are two matches witnessing that the property graph does not satisfy the constraint: The first match corresponds to the path $p_1 w_1 t_1 r_1 d_1 r_3 d_3$ (cf. Example 2.3), and the second one to the path $p_1 w_1 t_1 r_1 d_1 r_4 d_3$. The topological 1-neighbourhood errors are $O^1_1 = \{p_1, w_1, t_1\} \cup \{d_1, r_3, d_3\}$, and $O^1_2 = \{p_1, w_1, t_1\} \cup \{d_1, r_4, d_3\}$. In contrast to the (full) topological errors $O_1 = \{p_1, w_1, t_1, r_1, d_1, r_3, d_3\}$ and $O_2 = \{p_1, w_1, t_1, r_1, d_1, r_4, d_3\}$, the edge r_1 is missing from O^1_1 and O^1_2. Thus, when using O^1_1 and O^1_2 instead of O_1 and O_2, the repair obtained by deleting r_1 is not found. On the other hand, the errors are smaller, and in general their size is bounded linearly in k, and not in the size of the property graph.

5 EXPERIMENTS

To showcase that our approach is applicable in practice we conducted several experiments: We first assess the base pipeline in terms of runtimes and scalability with respect to the number of errors and constraints, and compare the performances and quality of the ILP and both greedy algorithms. We then study the effects of the optional steps of the repair pipeline on quality and runtime. Finally, we conduct a qualitative study aiming at studying the behaviour of the pipeline on a real-world dataset with real-world constraints. For the evaluation, we use the four property graphs of different sizes listed in Table 1. The top three are real-world property graphs, and the LDBC property graph is generated with the corresponding benchmark. We implemented our repair pipeline [43], illustrated in Figure 2, in Python 3.13. For solving (I)LPs, we use HiGHS [28]. As graph database system, we use Neo4j 5.26 (community edition). All experiments were run on a virtual machine with 8 physical cores of an Intel(R) Xeon(R) Silver 4214 CPU [17] and 125GiB RAM.

5.1 Assessing the Base Pipeline

In this section, we assess the performance and quality of the *base pipeline*, that is our repair pipeline without any optional steps. For this purpose, we generated constraints with pattern shapes as outlined in Figure 5 for the property graphs in Table 1. These pattern shapes cover most shapes that have been observed in real-life query logs [9] while also exhibiting recursion. Notably, the 1-way shape alone covers 74.61% of the valid path patterns observed [9], and the loop shapes include triangles, since the orange dashed shape can take the form of a pattern which matches paths of length 2. An instance of a 1-way pattern with 3 edge and 4 node patterns is $z = \left[(:\mathtt{a}) \xrightarrow{:\mathtt{c}} (:\mathtt{b}) \xrightarrow{:\mathtt{e}} (:\mathtt{d}) \xrightarrow{:\mathtt{f}} (:\mathtt{h})\right]^+$. A constraint then consists of a pattern combined with an empty filter and the condition {`false`}. This maximizes the number of errors for our scaling experiments. Moreover, to increase the number of constraints (and errors), we added, to each graph, an additional 10% of edges – labelled with a special label – randomly between nodes, which were already connected by an edge in the original graph, to preserve the topology. We emphasize that there is no correlation between the additional edges and errors: they can partake in an error or not, like other edges. Their sole purpose is to *increase* the number of paths, and

Table 1: Property graphs used for experiments, number $|\Sigma|$ of constraints, and number of errors

Name	#Nodes	#Edges	1-way		2-rep		2-way		loop		3-split	
			$\|\Sigma\|$	#errors	$\|\Sigma\|$	#errors	$\|\Sigma\|$	#errors	$\|\Sigma\|$	#errors	$\|\Sigma\|$	#errors
ICIJ Property Graph [29]	2 016 523	3 339 267	14	558799	19	397108	39	7553	188	56614	152	876933
Italian Legislative Graph [15, 16]	411 787	1 117 528	20	166977	–		53	443695	61	3783	–	
Coreutils Code Property Graph [47]	206 506	387 284	20	45728	–		52	3707	–		–	
LDBC, scaling factor 0.3 [1, 2]	940 322	5 003 201	5	79064	6	72756	–		56	423	4	2734499

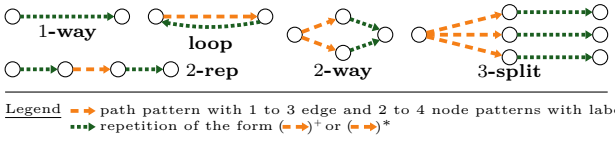


Figure 5: Shapes of RGPC patterns used in Sections 5.1 and 5.3

hence errors. The number of constraints and the total number of errors, for each shape and property graph, are listed in Table 1. Note that no ground truth is available. Instead, the given graphs are regarded as dirty, and our repairs are clean by definition. The repairs obtained by the ILP algorithm serve as best possible repairs.

We compare the performance and quality of the two greedy and ILP algorithms. We use runtimes[6] as performance, and the number of deletions as quality measure – a smaller number of deletions means a higher quality since it corresponds to less information loss. Note that measures like the F_1-score for quality are not meaningful: since our holistic approach always repairs all errors, there are no false negatives, and by definition a repair cannot contain false positives. Hence, the F_1-score is always 1 (or 0 in case of a timeout).

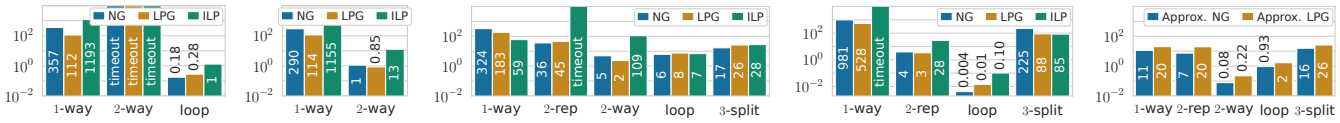
Figures 6a to 6d show the runtime of the naive greedy (NG), LP-guided greedy (LPG), and the ILP algorithms. Note that we used a log-scaled y-axis to obtain readable plots. For example, Figure 6a shows the performance of the base pipeline when run on the legislative property graph with the sets of 1-way, 2-way, and loop constraints outlined in Table 1. The ILP algorithm takes in general more time than the naive greedy algorithm, and as a trade-off, it deletes less objects[7], cf. Figures 7a to 7d (which also have a log-scaled y-axis, but a different range than the runtime plots). The LP-guided greedy achieves the same quality as the ILP algorithm in almost all cases (cf. Figures 7a to 7d). At the same time it competes with the naive greedy algorithm performance-wise. In many cases it even performs better. Inspecting the solutions of the LP-solver revealed that it converges towards integral solutions in these cases.

Overall, the experiments show that the LP-guided greedy algorithm yields higher-quality repairs with about 35% fewer deletions than the naive greedy, matching the quality of repairs yielded by the ILP algorithm. It also offers a runtime advantage of over 97% (Figure 6c, 2-way) compared to the ILP strategy, and beats the naive greedy algorithm in a case where the ILP one timed out (Figure 6d, 1-way). These observations in the general case are however not always true as demonstrated by Figure 6c (ICIJ): the ILP algorithm outperforms both greedy algorithms for 1-way constraints, which exhibit a high number of errors with paths containing many cycles.

Approximate Repairs. We now investigate approximate repairs and deepen the comparison between the naive and the LP-guided greedy algorithms by disabling the trimming phase in the pipeline. The results for the ICIJ graph are shown in Figures 6e and 7e. We observe that the selection phase of the naive greedy performs better than the one of the LP-guided greedy, and compared with the full runtimes in Figure 6c, the trimming phase is responsible for the majority of the runtime for both. In particular, the trimming phase of the naive greedy is responsible for making it perform worse overall. Correlating the number of proposed edge deletions indicates that this is because, for the naive greedy, more edge deletions are trimmed. When comparatively many edge deletions have to be trimmed for the LP-guided greedy, it can be outperformed by the naive greedy algorithm (e.g. ICIJ, 2-rep). Lastly, the selection phase of the LP-guided greedy algorithm can yield a good approximation or even a proper repair in the majority of cases, with a 89% reduction in runtime (1-way). Similar results for the other datasets are provided in the extended version [42, Appendix C].

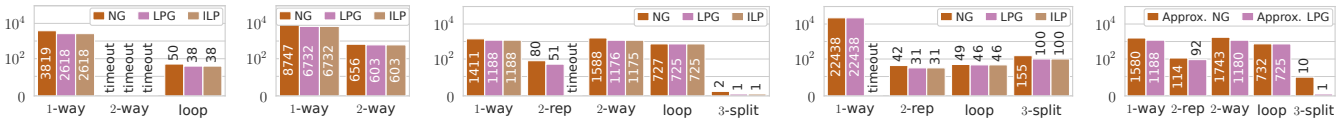
Scalability. As for the scalability of our pipeline with respect to the numbers of constraints and errors, Figures 6c and 6a attest that our pipeline can handle up to 188 loop constraints for the ICIJ graph, while it runs into a timeout for the smaller legislative graph with 53 2-way constraints. Regarding the size of errors, the maximal size we observed were 73 objects (Legislative, loop). Figure 6d shows that over 2.7 million errors due to 3-split constraint can be repaired comparatively fast, while 79064 errors due to 1-way constraints are challenging. Comparing the results for 1-way constraints for the ICIJ and LDBC graphs, we observe that our pipeline performs better for the former, even though there are fewer constraints (of the same shape) and fewer errors for the smaller LDBC graph (cf. Table 1, 1-way column). To investigate this case further, we ran the experiments for the ICIJ and LDBC graphs with subsets of 1-way constraints, and thus fewer errors. The resulting runtimes are shown in Figures 8a and 8b, and the number of deletions in Figures 8c and 8d. The y-axes in these plots are labelled with the numbers of errors. For the LDBC graph, compared to the number of errors, a high number of edges are deleted – up to 67% for the LP-guided greedy and ILP algorithms, and 73% for naive greedy algorithm – while only very few deletions are necessary to repair the ICIJ graph. This indicates that our repair pipeline performs well, if the repair can be achieved with a low number of deletions. Constraints and number of errors play a relatively negligible role.

[6]We do not measure error detection via queries, which heavily depends on the database system and is not the focus of our paper.
[7]Due to the nature of our constraints, the pipeline always opts for deleting edges without any optional feature enabled.



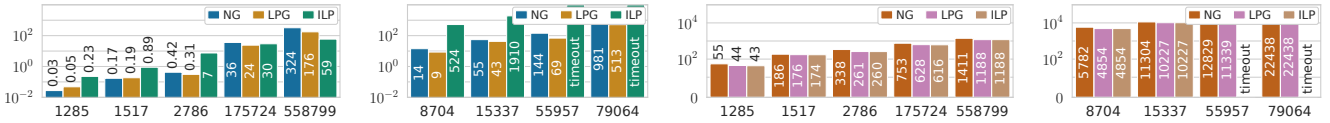
(a) Legislative (b) Coreutils (c) ICIJ (d) LDBC (e) ICIJ/Approximate

Figure 6: Runtimes in seconds of the naive greedy (NG), LP-guided greedy (LPG), and the ILP algorithms



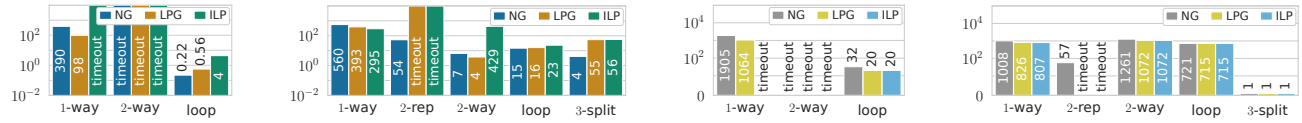
(a) Legislative (b) Coreutils (c) ICIJ (d) LDBC (e) ICIJ/Approximate

Figure 7: Number of edge deletions proposed by the naive greedy (NG), LP-guided greedy (LPG), and the ILP algorithms



(a) ICIJ/Runtimes/1-way (b) LDBC/Runtimes/1-way (c) ICIJ/Edge Deletions/1-way (d) LDBC/Edge Deletions/1-way

Figure 8: Results of the scaling experiments for the naive greedy (NG), LP-guided greedy (LPG), and the ILP algorithms



(a) Legislative/Runtimes (b) ICIJ/Runtimes (c) Legislative/Label Deletions (d) ICIJ/Label Deletions

Figure 9: Runtimes (in seconds) with Step 2 enabled and number of label deletions for the legislative and ICIJ graphs

5.2 Minimizing Deletions of Important Objects

By default our pipeline assigns uniform weights to edges, and node weights are determined by degree. To improve robustness, we allow the pipeline to incorporate varying *custom weights* $w_c(o) > 0$ for nodes and edges $o \in N \cup E$, stored as property values in the property graph, with the aim of reducing the deletion of objects deemed "important". Internally, the weight $w(e)$ of an edge is then $w_c(e)$, and the weight $w(n)$ of a node is $w_c(n)$ plus the sum of all $w(e)$, for all incident edges e. The objective of the pipeline is then shifted from minimizing the number of deletions to minimizing the reduction of the total custom weight across all nodes and edges.

To assess this feature, we used PageRank scores as weights for nodes and edges[8]. Without custom weights, the total PageRank value decreases by 0.09% (1-way constraints) when applying the LP-guided greedy algorithm on the ICIJ graph. With custom weights, the decrease is limited to 0.07% (2-way constraints), corresponding to a 24% reduction in weight loss, indicating that fewer high-weight edges are deleted. The number of deletions is shown in Figure 10d, and compared with Figure 7c, showing an increase ranging from 1% (2-rep constraints) to 43% (2-way constraints). The runtimes are shown in Figure 10c. Compared with the base case in Figure 6c, custom weights can have a positive or negative impact on runtime. More details are provided in the extended version [42, Appendix C].

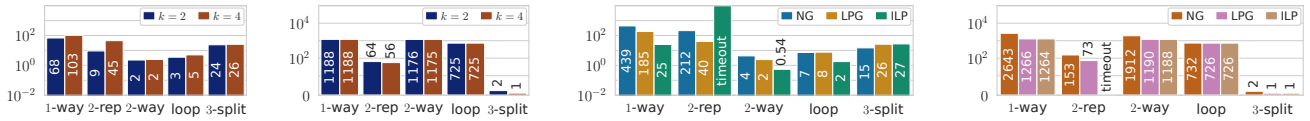
[8]Page rank for edges are computed using the line graph: Every edge becomes a node, and there is an edge (e_1, e_2) if there is a path $v_0 e_1 v_1 e_2 v_2$ in the original graph.

5.3 Label Removals and Neighbourhood Repairs

We now evaluate the optional steps of our pipeline, and compare them to the base pipeline established in Section 5.1.

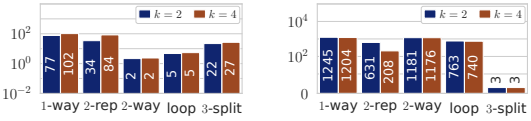
To show the effects of Step 2, we present the runtimes and number of label deletions for the legislative and the ICIJ graph in Figure 9. Recall that extending errors with essential tokens results in more, and larger errors. Consequently, the (I)LPs are also larger. Furthermore, there tend to be more equally weighted repairs: instead of deleting an edge, there is now the choice between deleting any label from the edge, or one of its endpoints. The runtimes increase considerably – by up to 500% (Figure 6c vs. Figure 9b, 1-way). The ILP algorithm even timed out in one more case (Figure 9a, 1-way).

However, comparing with Figures 7a and 7c, we observe a reduction of up to 50% of object deletions for the naive greedy algorithm (Legislative, 1-way), of up to 47% for the ILP algorithm (Legislative, loop), and of up to 59% for the LP-guided greedy (Legislative, 1-way, the ILP algorithm times out in this case, cf. Figure 9c).



(a) Step 3/Runtimes **(b) Step 3/Edge Deletions** **(c) Custom Weights/Runtimes** **(d) Custom Weights/Edge Deletions**

Figure 10: Results of the LP-guided greedy algorithm for neighbourhood (Step 3) and custom weight repairs for ICIJ



(a) ICIJ/Runtimes **(b) ICIJ/Edge Deletions**

Figure 11: LP-guided greedy algorithm on sampled errors

Notably, even the naive greedy algorithms can yield a repair with less label deletions than the ILP with (only) edge deletions (Legislative, 1-way). Note that the greedy algorithms can sometimes even be faster if label deletions are allowed which has to be attributed to the structure of the errors and the (I)LP (e.g. Figure 9a, 1-way).

A similar trend can be observed for the other datasets; for space reasons they are provided in the extended version [42, Appendix C].

With Step 3 we observe a runtime reduction of up to 62% compared to the base pipeline while maintaining almost the same quality, i.e. number of deletions. We present the runtimes and number of proposed edge deletions for ICIJ and neighbourhood $k = 2$ as well as $k = 4$ in Figures 10a and 10b, for the LP-guided greedy algorithm. Compared to the base case shown in Figure 6c, the performance is better in cases where the base pipeline takes a significant amount of time, i.e. more than a few seconds.

The numbers of deletions are quite similar for $k = 2$ and $k = 4$, and more importantly, the results for the base pipeline as in Figure 7c, except for the experiment for 2-rep constraints. That is, in all cases where matches consisting of *shortest paths* are (almost) completely covered by the k-endpoint neighbourhood.

To assess whether picking the neighbourhoods of path endpoints is a suitable strategy for reducing the size of errors, we performed the same experiments with random samples of errors: that is, we sampled $2k$ edges (and their endpoints) from each error. The result for the LP-guided greedy algorithm on the ICIJ graph is shown in Figure 11. Compared with the result for neighbourhood errors shown in Figure 10, the runtimes do not differ significantly, but the number of deletions can explode if paths are long (9.86x for 2-rep). This suggests that intersections of (topological) errors are indeed more likely to happen near path endpoints, and taking a random sample from a long path is less likely to contain these intersections.

Similar plots to the ones for Steps 2 and 3 for the other datasets are available in the extended version [42, Appendix C].

5.4 A Qualitative Study

To showcase that our repair pipeline is viable in practice, we conducted a qualitative study on the real-world ICIJ property graph, which is built and used by the International Consortium of Investigative Journalists. Since this property graph was assembled from several data sources, it contains some real-world inconsistencies, detected using the RGPC constraints in Table 2.

The constraints are concerned with *entities*, which model companies and organisations, and *officers*, which are persons involved with entities. We note that entities can also be in an `officer_of` relationship with other entities, although neither of them is labelled as an officer. Constraint γ_1 states that entities and/or officers in a `same_name_as` relationship, have indeed the same name. Constraint γ_2 is a typical denial constraint forbidding an entity from having an inactivation date, if it is still active. Constraint γ_3 models an *exclusivity* condition for entities [4]: no entity can have two *sole* directors. Finally, γ_4 ensures that a president of an entity cannot gain an advantage by indirectly influencing another entity they are associated with, through their presidency. The number of errors for each constraint is given in Column 3 of Table 2.

We first discuss the repairs yielded by the base pipeline with the ILP algorithm. The numbers of deletions are given in Column 4 of Table 2. For γ_1, `same_name_as` relationships are deleted. This preserves the original names stored in the properties (unifying names could lead to false accusations in this scenario). Since errors for this constraint cannot share edges, one deletion per error is expected. Constraint γ_2 is an example where the pipeline is forced to delete nodes, since each error consists of exactly one node. Thus, the nodes are deleted, and consequently so are all their 177 incident edges. For γ_3 one of the two relationships in each error is deleted, which shows that symmetries are resolved properly (the errors do not share any objects in this case). The repair for γ_4 shows that the number of deletions can be significantly less than the number of errors – our pipeline exploits that errors have edges in common.

We investigate robustness w.r.t. graph topology changes with γ_4, since it encompasses a non-trivial path pattern. Assuming that subpatterns describing paths of `same_name_as`- and `president_of`-labelled edges are important, we double the weight of these edges. The repair pipeline then proposes the deletion of 14 instead of 10 edges but preserves all of the aforementioned edges. It also preserves at least 168 of 203 matches of the first pattern of γ_4. Without the custom weights, it preserves only between 96 and 117 matches of this pattern.

When enabling label deletions, the graph can be repaired with 6323 label deletions, which is only about 35% of the 18255 node and edge deletions without it. 1394 labels are removed from edges, effectively deleting the concrete relationship but preserving the knowledge that there is some (potential) relationship.

Using the naive greedy algorithm instead of the ILP algorithm resulted in up to 2 more deletions for γ_4 (depending on the order in which the database returned errors), the same amount of deletions for the other constraints. The LP-guided greedy yielded the same number of deletions. With the ILP algorithm taking under 4 seconds

Table 2: Real-world like RGPC constraints for the ICIJ property graph

ID	RGPC constraint $q; F \Rightarrow C$	#errors	#deletions
γ_1	$(x{:}\texttt{Entity} \vee \texttt{Officer}) \xrightarrow{:\texttt{same_name_as}} (y{:}\texttt{Entity} \vee \texttt{Officer}); \emptyset \Rightarrow \{x.\texttt{name} = y.\texttt{name}\}$ ↪ All entities and officers in a same_name_as relationship have indeed the same name.	18000	18000 edges 0 nodes
γ_2	$(x{:}\texttt{Entity}); \{x.\texttt{inactivation_date} \leq \text{2025-07-01}\} \Rightarrow \{x.\texttt{status} \neq \text{“Active”}\}$ ↪ All entities with an inactivation date in the past do not have the status “Active”.	46	177 edges 46 nodes
γ_3	$(x{:}\texttt{Officer}) \xrightarrow{:\texttt{sole_director_of}} (y{:}\texttt{Entity}), (z{:}\texttt{Officer}) \xrightarrow{:\texttt{sole_director_of}} (y); \emptyset \Rightarrow \{x = z\}$ ↪ An entity cannot have more than one sole director.	44	22 edges 0 nodes
γ_4	$(x{:}\texttt{Officer}) \xrightarrow{:\texttt{president_of}} (y{:}\texttt{Entity}) \left[(:\texttt{Entity} \vee \texttt{Officer}) \xrightarrow{:\texttt{officer_of} \vee \texttt{same_name_as}}\right]^+ (z{:}\texttt{Entity}), (x) \xrightarrow{:\texttt{officer_of}} (z); \emptyset \Rightarrow \{y = z\}$ ↪ No president is indirectly in control of an entity, e.g. through a chain of shell companies, they are directly an officer of.	19	10 edges 0 nodes

for the set of all constraints, there is no advantage of using either greedy algorithm or neighbourhood errors in this scenario.

6 RELATED WORK

Error repairing for PG-Constraints has not been addressed in prior work besides the assessment that PG-Constraints can be rewritten into GQL queries for querying errors [4].

Error repairing for less expressive graph data models, such as labelled graphs, focuses on chase-based approaches [10, 11, 21, 22, 32, 39, 40], which iteratively add properties, or even edges to repair a graph. Thus, this repair mode is orthogonal to our delete-based approach, and consequently not comparable. In particular, insertion-only repair modes are not suitable for repairing violations of denial constraints in our examples and experimental settings [20].

A different approach for repairing graphs is with respect to neighbourhood constraints, that is, constraints which restrict the labels of a node depending on the labels of its neighbours [41]. Their repair mode involves changing labels, and, if that is not enough, changing the neighbourhood of a node, which is again incompatible with our repair mode. They introduce a greedy variant of their algorithm to strike a balance between runtime and repair quality.

User-centric repairs for property graphs have been studied, albeit w.r.t. constraints for less expressive graph data models [36, 37]. Their repair model focuses on label and property changes, but also allows the deletion of edges as fall-back. Letting users repair errors consisting of variable-length paths requires carefully crafted human-machine interfaces, which is beyond the scope of this work.

The idea of using vertex covers of conflict hypergraphs originates from theoretical work for repairing relational databases [12, 44], but these repairs do not involve dependencies between objects and arbitrary long paths. A notable consequence is that, unlike in the relation setting, not every minimal vertex cover corresponds to a repair, which we addressed by using minimum weighted vertex covers. We note that modelling dependencies explicitly (like foreign-key constraints) instead of using weights, yields an unreasonable amount of (artificial) errors in the case of property graphs, since they would involve all nodes, edges, and labels. For the vast literature on error repairing for other database models and repair modes, including the relational model, we refer to surveys [13, 19].

We note that the vertex cover problem for hypergraphs is even W[2]-hard [25], which means that it is very unlikely that there is an FPT algorithm. A common strategy for solving ILPs are LP-rounding algorithms [e.g., 35], which solve the LP variant of the ILP and then use *randomized rounding* to obtain an integral solution. Our LP-guided greedy algorithm uses a trivial rounding step instead, which is arguably sufficient, as shown by our experiments in Section 5.

Theoretical foundations for RGPC patterns have been investigated, in particular CRPQs with path variables [6, 23], which even allow for “regular” comparisons of paths. Notably, unlike RGPC patterns, CRPQs only match edge labels (i.e. no node labels), and do not support label expressions. The same differences apply to our RGPC automata model and automata constructed for CRPQs [6].

Lastly, we defined RGPC patterns as a syntactical fragment of GPC patterns to abstract core features of the pattern matching sub-language of GQL and SQL/PGQ [26]. A useful feature in practice and supported by these languages are patterns which allow for traversing edges in reverse direction. We outline how our pipeline can be extended with this feature and other differences in the extended version [42, Appendix A.3]. It is known that enumerating matches of GPC patterns is in polynomial space in the size of the property graph but not in polynomial space in the size of the pattern [26]; making the enumeration of all errors in large graphs intractable, and sets boundaries for repair procedures relying on enumerating all errors. Finally, PG-Constraints can capture dependencies based on regular path queries for which problems related to graph repair like certain query answering are undecidable [see, e.g. 7, 27, 38].

7 CONCLUSION AND FUTURE WORK

In this paper, we addressed error repairing for property graphs under PG-Constraints. We designed a repair pipeline for RGPC constraints with user-selectable trade-offs and demonstrated its effectiveness through extensive experiments.

As future work, we plan to study the practical limits of error repairing under PG-Constraints. Additional features could be incorporated such as advanced GQL pattern matching [18, 24]. The repair model may also be extended to allow property deletions, though this is challenging since deletions can introduce new errors. Other repair modes could combine our deletion-based approach with insertion techniques from the state of the art. Finally, given the large number of errors recursive constraints may generate, and inspired by work on incremental error detection in the relational setting [31], we aim to explore how errors can be efficiently prevented from being inserted in the first place.

ACKNOWLEDGMENTS

The authors were supported by ANR-21-CE48-0015 VeriGraph. A. Bonifati is also funded by IUF Endowed Chair.

REFERENCES

[1] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. https://doi.org/10.48550/arxiv.2001.02299 arXiv:2001.02299

[2] Renzo Angles, Peter A. Boncz, Josep Lluís Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martínez-Bazan, Venelin Kotsev, and Ioan Toma. 2014. The linked data benchmark council: a graph and RDF industry benchmarking effort. *ACM SIGMOD Record* 43, 1 (May 2014), 27–31. https://doi.org/10.1145/2627692.2627697

[3] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. 2023. PG-Schema: Schemas for Property Graphs. *Proceedings of the ACM on Management of Data* 1, 2 (June 2023), 1–25. https://doi.org/10.1145/3589778

[4] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, New York, NY, USA, 2423–2436. https://doi.org/10.1145/3448016.3457561

[5] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. 1999. Consistent Query Answers in Inconsistent Databases. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA (SIGMOD/PODS99)*, Victor Vianu and Christos H. Papadimitriou (Eds.). Association for Computing Machinery (ACM) Press, New York, NY, USA, 68–79. https://doi.org/10.1145/303976.303983

[6] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Transactions on Database Systems* 37, 4 (Dec. 2012), 1–46. https://doi.org/10.1145/2389241.2389250

[7] Catriel Beeri and Moshe Y. Vardi. 1981. The Implication Problem for Data Dependencies. In *Automata, Languages and Programming, 8th Colloquium, Acre (Akko), Israel, July 13-17, 1981, Proceedings* (1981) *(Lecture Notes in Computer Science)*, Shimon Even and Oded Kariv (Eds.), Vol. 115. Springer Berlin Heidelberg, Berlin, Heidelberg, 73–85. https://doi.org/10.1007/3-540-10843-2_7

[8] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. 2005. A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005 (SIGMOD-/PODS05)*, Fatma Özcan (Ed.). Association for Computing Machinery (ACM), New York, NY, USA, 143–154. https://doi.org/10.1145/1066157.1066175

[9] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An Analytical Study of Large SPARQL Query Logs. *The VLDB Journal* 29, 2–3 (May 2020), 655–679. https://doi.org/10.1007/s00778-019-00558-9

[10] Yurong Cheng, Lei Chen, Ye Yuan, and Guoren Wang. 2018. Rule-Based Graph Repairing: Semantic and Efficient Repairing Methods. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 773–784. https://doi.org/10.1109/ICDE.2018.00075

[11] Yurong Cheng, Lei Chen, Ye Yuan, Guoren Wang, Boyang Li, and Fusheng Jin. 2022. Strict and Flexible Rule-Based Graph Repairing. *IEEE Transactions on Knowledge and Data Engineering* 34, 7 (2022), 3521–3535. https://doi.org/10.1109/TKDE.2020.3019817

[12] Jan Chomicki and Jerzy Marcinkowski. 2005. Minimal-Change Integrity Maintenance Using Tuple Deletions. *Information and Computation* 197, 1–2 (Feb. 2005), 90–121. https://doi.org/10.1016/J.IC.2004.04.007

[13] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data Cleaning: Overview and Emerging Challenges. In *Proceedings of the 2016 International Conference on Management of Data* (2016-06) *(SIGMOD/PODS'16)*. Association for Computing Machinery (ACM), San Francisco California USA, 2201–2206. https://doi.org/10.1145/2882903.2912574

[14] Alessandro Cimatti, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2011. From Sequential Extended Regular Expressions to NFA with Symbolic Labels. In *Implementation and Application of Automata*, Michael Domaratzki and Kai Salomaa (Eds.). Vol. 6482. Springer Berlin Heidelberg, Berlin, Heidelberg, 87–94. https://doi.org/10.1007/978-3-642-18098-9_10

[15] Andrea Colombo, Anna Bernasconi, and Stefano Ceri. 2024. *Italian Legislative Property Graph*. Zenodo. https://doi.org/10.5281/zenodo.11210265

[16] Andrea Colombo, Anna Bernasconi, and Stefano Ceri. 2024. Modelling Legislative Systems into Property Graphs to Enable Advanced Pattern Detection. https://doi.org/10.48550/ARXIV.2406.14935 arXiv:2406.14935

[17] Intel Coporation. 2026. Specification of the Intel(R) Xeon(R) Silver 4214 Processor. https://www.intel.com/content/www/us/en/products/sku/193385/intel-xeon-silver-4214-processor-16-5m-cache-2-20-ghz/specifications.html. Accessed: 2026-01-26.

[18] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar Van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data* (2022-06) *(SIGMOD/PODS '22)*. Association for Computing Machinery (ACM), Philadelphia PA USA, 2246–2258. https://doi.org/10.1145/3514221.3526057

[19] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 19, 1 (Jan. 2007), 1–16. https://doi.org/10.1109/TKDE.2007.250581

[20] Wenfei Fan. 2008. Dependencies Revisited for Improving Data Quality. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada (SIGMOD/PODS '08)*, Maurizio Lenzerini and Domenico Lembo (Eds.). Association for Computing Machinery (ACM), New York, NY, USA, 159–170. https://doi.org/10.1145/1376916.1376940

[21] Wenfei Fan, Wenzhi Fu, Ruochun Jin, Muyang Liu, Ping Lu, and Chao Tian. 2023. Making It Tractable to Catch Duplicates and Conflicts in Graphs. *Proceedings of the ACM on Management of Data* 1, 1 (May 2023), 1–28. https://doi.org/10.1145/3588940

[22] Wenfei Fan, Ping Lu, Chao Tian, and Jingren Zhou. 2019. Deducing Certain Fixes to Graphs. *Proceedings of the VLDB Endowment* 12, 7 (March 2019), 752–765. https://doi.org/10.14778/3317315.3317318

[23] Diego Figueira and Varun Ramanathan. 2022. When is the Evaluation of Extended CRPQ Tractable?. In *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022 (SIGMOD/PODS '22)*, Leonid Libkin and Pablo Barceló (Eds.). Association for Computing Machinery, New York, NY, USA, 203–212. https://doi.org/10.1145/3517804.3524167

[24] Diego Figueira and Miguel Romero. 2023. Conjunctive Regular Path Queries under Injective Semantics. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (2023-06) *(SIGMOD/PODS '23)*. Association for Computing Machinery (ACM), Seattle WA USA, 231–240. https://doi.org/10.1145/3584372.3588664

[25] Jörg Flum and Martin Grohe. 2006. *Parameterized Complexity Theory*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/3-540-29953-X

[26] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. GPC: A Pattern Calculus for Property Graphs. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (2023-06) *(SIGMOD/PODS '23)*. Association for Computing Machinery (ACM), Seattle WA USA, 241–250. https://doi.org/10.1145/3584372.3588662

[27] Nadime Francis and Leonid Libkin. 2017. Schema Mappings for Data Graphs. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (2017-05) *(SIGMOD/PODS'17)*. Association for Computing Machinery (ACM), Chicago Illinois USA, 389–401. https://doi.org/10.1145/3034786.3056113

[28] Qi Huangfu and J. A. J. Hall. 2018. Parallelizing the Dual Revised Simplex Method. *Mathematical Programming Computation* 10, 1 (Dec. 2018), 119–142. https://doi.org/10.1007/S12532-017-0130-5 https://highs.dev/.

[29] International Consortium of Investigative Journalists. 2026. ICIJ Offshore Leaks Database. https://offshoreleaks.icij.org/pages/database. Accessed: 2026-01-26.

[30] ISO Central Secretary. 2024. *Information Technology — Database Languages — GQL* (1 ed.). Standard ISO/IEC 39075:2024. International Organization for Standardization, Geneva, CH.

[31] Youri Kaminsky, Eduardo H. M. Pena, and Felix Naumann. 2024. Incremental Detection of Denial Constraint Violations. *Proceedings of the VLDB Endowment* 18, 4 (Dec. 2024), 1000–1012. https://doi.org/10.14778/3717755.3717761

[32] Selasi Kwashie, Lin Liu, Jixue Liu, Markus Stumptner, Jiuyong Li, and Lujing Yang. 2019. Certus: An Effective Entity Resolution Approach with Graph Differential Dependencies (GDDs). *Proceedings of the VLDB Endowment* 12, 6 (Feb. 2019), 653–666. https://doi.org/10.14778/3311880.3311883

[33] Yuxi Liu, Fangzhu Shen, Kushagra Ghosh, Amir Gilad, Benny Kimelfeld, and Sudeepa Roy. 2024. The Cost of Representation by Subset Repairs. *Proceedings of the VLDB Endowment* 18, 2 (Oct. 2024), 475–487. https://doi.org/10.14778/3705829.3705860

[34] Andrei Lopatenko and Leopoldo E. Bertossi. 2007. Complexity of Consistent Query Answering in Databases Under Cardinality-Based and Incremental Repair Semantics. In *Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, January 10-12, 2007, Proceedings (Lecture Notes in Computer Science)*, Thomas Schwentick and Dan Suciu (Eds.), Vol. 4353. Springer Berlin Heidelberg, Berlin, Heidelberg, 179–193. https://doi.org/10.1007/11965893_13

[35] Mourad El Ouali, Helena Fohlin, and Anand Srivastav. 2014. A randomised approximation algorithm for the hitting set problem. *Theoretical Computer*

Science 555 (Oct. 2014), 23–34. https://doi.org/10.1016/J.TCS.2014.03.029

[36] Amedeo Pachera, Angela Bonifati, and Andrea Mauri. 2025. Grafixer: Enabling User-Centric Repairs for Property Graphs. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, New York, NY, USA, 199–202. https://doi.org/10.1145/3722212.3725105

[37] Amedeo Pachera, Angela Bonifati, and Andrea Mauri. 2025. User-Centric Property Graph Repairs. *Proceedings of the ACM on Management of Data* 3, 1 (Feb. 2025), 85:1–85:27. https://doi.org/10.1145/3709735

[38] Sylvain Salvati and Sophie Tison. 2024. Containment of Regular Path Queries Under Path Constraints, In 27th International Conference on Database Theory, ICDT 2024, Paestum, Italy, March 25-28, 2024 (2024), Graham Cormode and Michael Shekelyan (Eds.). *LIPIcs, Volume 290, ICDT 2024* 290, 17:1–17:19. https://doi.org/10.4230/LIPICS.ICDT.2024.17

[39] Larissa Capobianco Shimomura, George Fletcher, and Nikolay Yakovets. 2020. GGDs: Graph Generating Dependencies. In *The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020 (CIKM '20)*, Mathieu d'Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux (Eds.). Association for Computing Machinery, New York, NY, USA, 2217–2220. https://doi.org/10.1145/3340531.3412149

[40] Larissa Capobianco Shimomura, Nikolay Yakovets, and George Fletcher. 2024. Reasoning on property graphs with graph generating dependencies. *Information Sciences* 672 (2024), 120675. https://doi.org/10.1016/J.INS.2024.120675

[41] Shaoxu Song, Boge Liu, Hong Cheng, Jeffrey Xu Yu, and Lei Chen. 2017. Graph Repairing under Neighborhood Constraints. *The VLDB Journal* 26, 5 (May 2017), 611–635. https://doi.org/10.1007/S00778-017-0466-5

[42] Christopher Spinrath, Angela Bonifati, and Rachid Echahed. 2026. Repairing Property Graphs under PG-Constraints. https://doi.org/10.48550/arXiv.2602.05503 arXiv:2602.05503

[43] Christopher Spinrath, Angela Bonifati, and Rachid Echahed. 2026. *Repairing Property Graphs under PG-Constraints [Artifacts]*. Zenodo. https://doi.org/10.5281/zenodo.18301604

[44] Sławek Staworko, Jan Chomicki, and Jerzy Marcinkowski. 2012. Prioritized Repairing and Consistent Query Answering in Relational Databases. *Annals of Mathematics and Artificial Intelligence* 64, 2–3 (March 2012), 209–246. https://doi.org/10.1007/s10472-012-9288-8

[45] Jef Wijsen. 2005. Database repairing using updates. *ACM Transactions on Database Systems* 30, 3 (Sept. 2005), 722–768. https://doi.org/10.1145/1093382.1093385

[46] Mingyu Xiao, Sen Huang, and Xiaoyu Chen. 2024. Maximum Weighted Independent Set: Effective Reductions and Fast Algorithms on Sparse Graphs. *Algorithmica* 86, 5 (May 2024), 1293–1334. https://doi.org/10.1007/s00453-023-01197-x

[47] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy* (2014-05). IEEE, San Jose, CA, 590–604. https://doi.org/10.1109/SP.2014.44