SN Computer Science (2023) 4:775
https://doi.org/10.1007/s42979-023-02142-z

ORIGINAL RESEARCH

Checking Pattern Query Containment Under Shape Expression

Haruna Fujimoto[1] · **Nobutaka Suzuki**[1] · **Yeondae Kwon**[2]

Received: 28 March 2023 / Accepted: 12 July 2023
© The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd 2023

Abstract
Shape expression (ShEx) is a novel schema language for RDF/graph data and is already in use in various domains. For various kinds of schema and data including ShEx and RDF graph, query containment is one of the major fundamental problems and related to many important applications, e.g., determining independence of queries from updates and rewriting queries using views. In particular, ShEx is a schema language that defines the data structure of graphs, and thus considering query containment from the perspective of graph structure under such a schema language is intrinsically an interesting problem. In this paper, we consider a query containment problem under ShEx, where a query is defined as a pattern graph with projection. We adopt a graph-theoretic approach to deal with the containment problem, and propose a new algorithm for solving the problem. In our experiments, we first verified that the results of our algorithm were correct for all the examined queries. We also show that ShEx types effectively reduce the search space for checking pattern query containment.

Keywords RDF · Query containment · Graph data

Introduction

Over the years, RDF/graph data have been used in various fields. For various kinds of data including RDF/graph, query containment is one of the major fundamental problems. Here, query containment is the problem of determining if the result of a query is always included in the result of another query. In addition to being theoretically interesting in its own right, query containment is related to many important applications: query optimization, determining independence of queries from updates, and rewriting queries using views.

In this paper, we consider a query containment problem under Shape Expression (ShEx). Here, ShEx is a novel schema language for RDF/graph data, which is being steered by the Shape Expression Community Group. ShEx is designed for capturing *structural* features of RDF/graph data. An ShEx schema assigns types to the nodes of an RDF/graph data and allows defining a set of types that impose structural constraints on nodes and their immediate neighbors using a regular bag expression (RBE) [2]. ShEx is applicable in multiple contexts, e.g., model development, legacy review, and model documentation [3]. In addition, large amount of data are being generated and exchanged in some advanced areas, e.g., smart cities [4, 5], and studies are being made to manage such data as RDF [6, 7]. ShEx can also be applied to such areas.

RDF Schema (RDFS) has long been known and used as a schema language for RDF. However, RDFS is actually an ontology description language rather than a structural schema definition language [2]. As the application of RDF/graph data becomes more widespread, the need for schema languages for specifying the *structural* aspects of

This article is part of the topical collection "Advances on Web Information Systems and Technologies" guest edited by Joaquim Filipe, Francisco José Domínguez Mayo and Massimo Marchiori.

An earlier version of this paper, Fujimoto and Suzuki [1], was presented in the 18th International Conference on Web and Information Systems and Technologies (WEBIST 2022). We have elaborated the explanation throughout this paper and also updated the evaluation experiments.

✉ Nobutaka Suzuki
nsuzuki@slis.tsukuba.ac.jp

Haruna Fujimoto
s2121649@klis.tsukuba.ac.jp

Yeondae Kwon
kwony518@affrc.go.jp

[1] University of Tsukuba, 1-2 Kasuga, Tsukuba, Ibaraki 3058550, Japan

[2] Research Center for Agricultural Information Technology, NARO, 2-14-1 Nishi-Shinbashi, Minato-ku, Tokyo 1050003, Japan

Published online: 12 October 2023

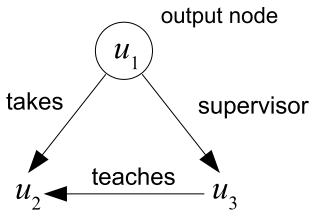

Fig. 1 Example of pattern query with projection

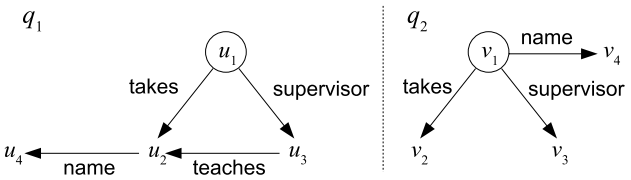

Fig. 2 Queries q_1 and q_2

such data increases. With this background, new schema languages, such as Shape Expression (ShEx) and Shapes Constraint Language (SHACL), have been proposed. ShEx shares many fundamental features with Shapes Constraint Language (SHACL) [8]; thus, the results of this paper can also be applied to SHACL.

As for query language, we focus on pattern graph with projection. For example, Fig. 1 depicts a tiny example consisting of three nodes in which only the value of circled node u_1 is an output. Intuitively, u_1, u_2, u_3 stand for a student, course, and professor, respectively, and thus, this query outputs any student taking a course taught by his/her supervisor. Consider solving the containment problem for such a query. If neither projection nor schema is considered, the problem is equivalent to subgraph isomorphism; q_1 is contained in q_2 if and only if q_2 is a subgraph of q_1. This no longer holds; however, if projection is allowed and schema is presented. That is, even if q_1 is not a subgraph of q_2 and vice versa, one of q_1 and q_2 may contain the other. For example, consider Fig. 2, where u_1, v_1 are student nodes, u_2, v_2 are course nodes, and u_3, v_3 are professor nodes. Suppose that schema S asserts that "name" is mandatory for students and courses, and that q_1, q_2 are queries under S. Then, whether $q_1 \not\subseteq q_2$ holds depends on S; $q_1 \not\subseteq q_2$ if the projections and S are ignored, $q_1 \subseteq q_2$ otherwise.

To deal with this problem, we devised a novel simple algorithm for checking pattern query containment under ShEx schema. For given pattern queries q_1 and q_2, the algorithm first finds a correspondence between the nodes of q_1 and q_2, which is obtained from a maximum common subgraph of q_1 and q_2. This problem is NP-hard, but the running time can be reduced using the ShEx types which can narrow the search space. Based on the correspondence, we check if there is an edge e in q_2 but not in q_1, such that e affects the

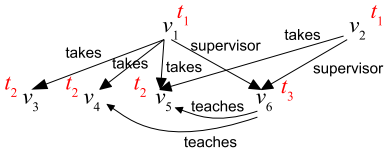

Fig. 3 Example of valid graph G

query containment w.r.t. q_1. If there is no such edge in q_2, then the algorithm concludes that q_1 is contained in q_2. We show the soundness of the algorithm. In our experiments, we generated hundreds of pairs of queries, and we verified that the results of our algorithm were correct for all the pairs of the generated queries. We also showed that ShEx types can be used to reduce the search space for checking pattern query containment.

The rest of this paper is organized as follows. Section "Preliminaries" gives preliminary definitions. Section "Algorithm for checking query containment" describes an algorithm for checking query containment under ShEx. Section "Experiments" presents experimental results. Section "Related work" presents the related works. Section "Conclusion" summarizes this paper.

Preliminaries

Let Σ be a set of labels. A *labeled directed graph* (*graph* for short) over Σ is denoted $G = (V, E)$, where V is a set of nodes and $E \subseteq V \times \Sigma \times V$ is a set of *edges*. An edge labeled by l from a node v to a node v' is denoted (v, l, v'). A *pattern graph* (or *query*) is denoted $q = (V(q), E(q), P)$, where $(V(q), E(q))$ is a graph and P is a tuple of *output nodes*. For example, the query in Fig. 1 is denoted $(V(q), E(q), P)$, where

$$V(q) = \{u_1, u_2, u_3\},$$
$$E(q) = \{(u_1, \text{supervisor}, u_3), (u_1, \text{takes}, u_2), (u_3, \text{teaches}, u_2)\},$$
$$P = (u_1).$$

By $\text{Ans}(q, G)$, we mean the set of answer tuples of q over G. For example, consider the pattern query q in Fig. 1 and the graph G in Fig. 3. Then, $\text{Ans}(q, G) = \{(v_1), (v_2)\}$.

The content model of type in ShEx can be modeled as regular bag expression (RBE) [2]. RBE is defined similarly to regular expression except that RBE uses *unordered* concatenation instead of ordered concatenation. Let Γ be a set of types. Then, RBE over $\Sigma \times \Gamma$ is recursively defined as follows.

- ϵ and $a::t \in \Sigma \times \Gamma$ are RBEs. ϵ denotes "empty bag" having 0 occurrences of any symbol.

- If $r_1, r_2, \ldots, r_k$ are RBEs, then $r_1|r_2|\cdots|r_k$ is an RBE, where | denotes disjunction.
- If $r_1, r_2, \ldots, r_k$ are RBEs, then $r_1 \parallel r_2 \parallel \cdots \parallel r_k$ is an RBE, where $\parallel$ denotes unordered concatenation.
- If r is an RBE, then r^*, r^+, and $r?$ are RBEs. Here, '$*$' indicates zero or more repetitions of r, $r^+ = r \parallel r^*$, and $r? = \epsilon|r$.

For example, let $r = (a::t_1|b::t_2) \parallel c::t_3$ be an RBE. Since $\parallel$ is unordered, r matches not only $a::t_1\ c::t_3$ and $b::t_2\ c::t_3$ but also $c::t_3\ a::t_1$ and $c::t_3\ b::t_2$. In the following, we assume that any RBE is *single occurrence*, i.e., for any $a::t \in \Sigma \times \Gamma$ and any RBE r, $a::t$ occurs at most once in r.

A *ShEx schema* is denoted $S = (\Sigma, \Gamma, \delta)$, where Γ is a set of *types* and δ is a function from Γ to the set of RBEs over $\Sigma \times \Gamma$. For example, let $S = (\Sigma, \Gamma, \delta)$ be a ShEx schema, where $\Sigma = \{\text{takes}, \text{supervisor}, \text{teaches}\}$, $\Gamma = \{t_1, t_2, t_3\}$, and

$$\delta(t_1) = (\text{takes}::t_2)^* \parallel (\text{supervisor}::t_3)?,$$
$$\delta(t_2) = \epsilon,$$
$$\delta(t_3) = (\text{teaches}::t_2)^*.$$

In RBE, $a::t$ matches an edge e if the label of e is a and the target node of e is of type t. Thus, assuming that each node in Fig. 3 is of the type colored in red, the type of each node v_i matches the outgoing edges of v_i. Thus G is a valid graph of S.

For queries q_1, q_2, and an ShEx schema S, q_2 contains q_1 over S if for any valid graph G of S, $\text{Ans}(q_1, G) \subseteq \text{Ans}(q_2, G)$.

Algorithm for Checking Query Containment

Our algorithm is based on the node correspondence between q_1 and q_2. Such a correspondence may already be known in some cases, e.g., when comparing an updated query and its original one, we can easily identify the correspondence between their nodes. However, this is not always the case. Thus, our algorithm:

1. firstly finds node correspondences between q_1 and q_2 ("Finding node correspondence") if necessary, and then
2. Under the obtained node correspondences, the algorithm checks the containment of q_1 and q_2 ("Checking containment").

Thus, if we already know the node correspondence between q_1 and q_2, then we use the algorithm of "Checking containment" directly, otherwise we first use the algorithm of "Finding node correspondence" and then the one of "Checking containment".

Finding Node Correspondence

We assume that the size of output tuples of q_1 and q_2 are identical (otherwise, q_1 and q_2 are incomparable), and thus, we can identify the correspondence between the output nodes of q_1 and q_2. Thus, in the following, we consider finding correspondences of between their non-output nodes.

Node correspondence is expressed by function $\mu()$. Let $u \in V(q_1)$. We write $\mu(u) = v$ (and we also write $\mu(v) = u$) if u corresponds to $v \in V(q_2)$, and $\mu(u) = v_{\text{nil}}$ if there is no node corresponding to u, where v_{nil} is a new node not in $V(q_2)$.

Algorithm 1 finds the "maximum" correspondence(s) between q_1 and q_2. We first identify the type of each node in q_1 and q_2 under ShEx schema S (lines 1 and 2). This is done by the algorithm for checking satisfiability of pattern queries [9], since the type of each node is identified during checking satisfiability. We check node correspondence in order of distance from the output node(s) using queue Q introduced in line 3. In line 5, since u can correspond to only a node in q_2 of the same type, $T(u)$ collects such nodes for u. In lines 8–10, FindCandidates checks whether u corresponds to v in $T(u)$,[1] and then, traverse descendants of u and v recursively to find their correspondences. Finally, among the obtained correspondences in C, we find maximum ones. This is done by, for each correspondence μ in C, computing a maximum common edge subgraph of q_1 and q_2 under μ and choosing maximum ones among them (lines 11–23).

Algorithm 1 FindNodeCorrespondences

Input: ShEx schema $S = (\Sigma, \Gamma, \delta)$, queries q_1, q_2
Output: set of node correspondences

```
1: λ1 ← FindNodeType(q1, S)
2: λ2 ← FindNodeType(q2, S)
3: Let Q be a queue. Add Q the nodes in q1 adjacent to/from some output node.
4: Get a node u from Q
5: T(u) ← {v | v ∈ V(q2), λ1(u) = λ2(v)}
6: C ← ∅
7: μ ← nil
8: for each v ∈ T(u) ∪ {v_nil} do
9:     C ← C ∪ FindCandidates(q1, q2, Q, u, v, μ)
10: end for
11: max ← 0, M_max ← ∅
12: A1 ← CreateAdjacencyMatrix(q1)
13: for each μ ∈ C do
14:     A2 ← CreateAdjacencyMatrix(q2, μ)
15:     n ← CountCommonEdge(A1, A2)
16:     if n > max then
17:         max ← n
18:         M_max ← {μ}
19:     else if n = max then
20:         M_max ← M_max ∪ {μ}
21:     end if
22: end for
23: return M_max
```

[1] In line 8, we have "dummy" node v_{nil} as well as $T(u)$ in case that $T(u) = \emptyset$. In line 9, *copies* of Q and μ are passed into FindCandidates.

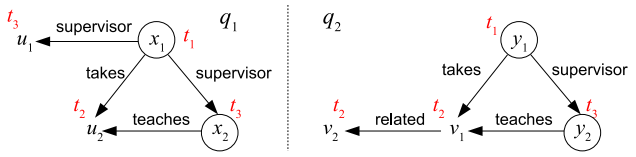

Fig. 4 Queries q_1 and q_2

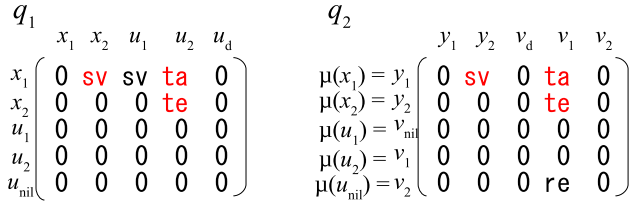

Fig. 5 Adjacency matrices of q_1 and q_2

Algorithm 2 FindCandidates

Input: queries q_1, q_2, queue Q, nodes $u \in V(q_1), v \in V(q_2)$, node correspondence μ
Output: set of node correspondences between q_1 and q_2

1: **if** $Adj(q_1, u) \cap \mu(Adj(q_2, v)) \neq \emptyset$ **then**
2: $\mu(u) \leftarrow v$
3: **else**
4: $\mu(u) \leftarrow v_{nil}$
5: **end if**
6: $N \leftarrow \{u' \in Adj(q_1, u) \mid u'$ is neither visited yet nor an output node$\}$
7: Add the nodes in N to Q
8: **if** Q is not empty **then**
9: Get a node u' from Q
10: **else** // all nodes are visited
11: **return** $\{\mu\}$
12: **end if**
13: $C(u') \leftarrow \{v \mid v \in T(u'), \forall u \in V(q_1)\, \mu(u) \neq v\}$
14: $C \leftarrow \emptyset$
15: **for each** $v' \in C(u') \cup \{v_{nil}\}$ **do**
16: $C \leftarrow C \cup$ FindCandidates$(q_1, q_2, Q, u', v', \mu)$
17: **end for**
18: **return** C

For a node u in q_1 and v in q_2, FindCandidates recursively traverses the descendants of u and v and finds correspondences between the nodes (Algorithm 2). For a query $q = (V(q), E(q), P)$, by $\text{Adj}(q, u)$, we mean the set of nodes adjacent to/from u in q. More precisely

$$\text{Adj}(q,u) = \{u' \in V(q) \mid (u,l,u') \in E(q) \text{ or } (u',l,u) \in E(q) \text{ for some } l \text{ and } u\}.$$

First, in lines 1–5, we check whether u should correspond to v or not. This is done by checking whether u and v have a "common" adjacent node, where $\mu(\text{Adj}(q_2, v)) = \{\mu(v') \mid v' \in \text{Adj}(q_2, v)\}$. If there is such a node, then $\mu(u)$ is set to v; otherwise, $\mu(u)$ is set to v_{nil}. In lines 6, we collect the nodes adjacent to u that are not visited yet. If such nodes are found, then the nodes are added to Q in line 7. If Q is not empty, then we get the next node u' from Q (line 9). Otherwise, μ is returned as the result, since all the nodes are visited. Finally, for each node v' in q_2, such that v' is of the same type as u' and that v' is not correspond to any node in q_1 yet, call FindCandidates to traverse the descendants of u' and v' in lines 16.

For example, consider queries q_1 and q_2 shown in Fig. 4, and suppose that the type of each node is obtained as shown in the figure. By the assumption, output nodes x_1 and x_2 of q_1 correspond to y_1 and y_2 of q_2, respectively, but no correspondence for the other nodes is known. In step 3 of Algorithm 1, u_1 and u_2 are added to Q. Suppose that u_2 is taken from Q in line 4. Since $\lambda_1(u_2) = t_2$, we have $T(u_2) = \{v_1, v_2\}$ in line 5. Suppose that FindCandidates is called with $u = u_2$ and $v = v_1$ in line 9. In line 1 of FindCandidates, $\text{Adj}(q_1, u_2) \cap \mu(\text{Adj}(q_2, v_1)) = \{x_1, x_2\}$; thus, we have $\mu(u_2) = v_1$ in line 2. In lines 6 and 7, no nodes are added to Q. In line 9, u_1 is taken from Q. Then, $C(u_1) = \emptyset$ in line 13 and FindCandidates is called with $u = u_1$ and $v = v_{\text{nil}}$. Then, $\mu(u_1)$ is set to v_{nil} in line 4, and since Q is empty μ is returned in line 11. Thus, we have $\mu(x_1) = y_1$, $\mu(x_2) = y_2$, $\mu(u_2) = v_1$, and $\mu(u_1) = v_{\text{nil}}$. Based on this correspondence, in lines 12 and 14 of Algorithm 1, we create adjacency matrices of q_1 and q_2, as shown in Fig. 5. There are three elements (colored red) appearing in both matrices at the same position, meaning that we have three common edges between q_1 and q_2 under the correspondence. Thus, we have $n = 3$ and $M_{max} = \{\mu\}$.

Checking Containment

Let G be a graph. By $u(G)$, we mean the undirected graph obtained by replacing each directed edge of G with an undirected one. A subgraph G' of G is *weakly biconnected* if any one node in $u(G')$ is removed, and the resulting undirected subgraph remains connected. A subgraph G' of G is *weakly biconnected component* if G' is a maximal weakly biconnected subgraph. For an edge $e = (v, a, v')$ in q_2, $(\mu(v), a, \mu(v'))$ is called the *corresponding edge* of e in q_1.

For queries q_1, q_2 and a ShEx schema S, we check if $q_1 \subseteq q_2$ as follows:

1. For each edge e in q_2, if e is not "answer-reducing" for q_1 and its corresponding edge e' is not in q_1, then add e' to q_1. Here, an "answer-reducing" edge is an edge, such that adding its corresponding edge to q_1 reduces the answer of q_1; in other words, the answer of q_1 is not preserved.
2. If q_2 is a subgraph of q_1, then return "true" (i.e., $q_1 \subseteq q_2$), otherwise return "false".

We show the idea of "answer-reducing" edge with an example. Let q_1, q_2 be queries shown in Fig. 6 with $\mu(v_1) = u_1$, $\mu(v_2) = u_2$, $\mu(v_3) = u_3$, $\mu(v_4)$ and $\mu(v_5)$ are new nodes. Let $S = (\Sigma, \Gamma, \delta)$ be a ShEx schema, where $\Gamma = \{t_1, t_2, t_3\}$ and δ is defined as follows:

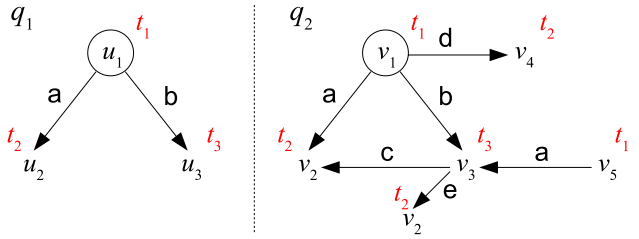

Fig. 6 Queries q_1 and q_2. q_2 contains three answer-reducing edges and one non answer-reducing one

$$\delta(t_1) = a::t_2 \parallel (b::t_3)? \parallel d::t_2,$$
$$\delta(t_2) = \epsilon,$$
$$\delta(t_3) = c::t_2 \parallel (e::t_2)?.$$

Then, (v_1, d, v_4) is *not* answer-reducing, as any node matched by u_1 must have an edge labeled by d under any valid graph of S. Thus, we can safely add (u_1, d, u_4) to q_1 without reducing the answer of q_1. On the other hand, (v_5, a, v_3) is answer-reducing, since (v_5, a, v_3) imposes an additional constraint that v_3 must be referenced by some edge labeled by a, meaning that adding (u_5, a, u_3) *reduces* the answer of q_1. Then, (v_3, c, v_2) is also answer-reducing, as adding (u_3, c, u_2) to q_1 yields a new weakly biconnected component (the triangle consisting of u_1, u_2, u_3), which imposes an extra constraint on q_1 that u_2 and u_3 must be connected by an edge labeled by c. Moreover, (v_3, e, v_2) is answer-reducing, as in $\delta(t_3)$ $e::t_2$ is qualified by ?, meaning that every node of type t_3 does not have an edge labeled by e. In Fig. 6, a query obtained by adding (u_1, d, u_4) to q_1 does not contain q_2 as a subgraph; thus, our algorithm concludes that $q_1 \not\subseteq q_2$.

To define answer-reducing edge formally, we also need min/max occurrences of label-type pair $a::t$ in an RBE. Let r be an RBE over $\Sigma \times \Gamma$ and $a::t \in \Sigma::\Gamma$. The *minimum occurrence* and *maximum occurrence* of $a::t$, denoted $\text{minocc}(r, a::t)$ and $\text{maxocc}(r, a::t)$, respectively, are defined as follows.

- If $r = a::t$, then $\text{minocc}(r, a::t) = \text{maxocc}(r, a::t) = 1$.
- If $r = r'^*$ and $a::t$ is in r', then $\text{minocc}(r, a::t) = 0$ and $\text{maxocc}(r, a::t) = \infty$.
- If $r = r'^+$ and $a::t$ is in r', then $\text{minocc}(r, a::t) = \text{minocc}(r', a::t)$ and $\text{maxocc}(r, a::t) = \infty$.
- If $r = r'?$ and $a::t$ is in r', then $\text{minocc}(r, a::t) = 0$ and $\text{maxocc}(r, a::t) = \text{maxocc}(r', a::t)$.
- If $r = r_1 | r_2 | \cdots | r_n$ or $r = r_1 \parallel r_2 \parallel \cdots \parallel r_n$, and $a::t$ is in r_i, then $\text{minocc}(r, a::t) = \text{minocc}(r_i, a::t)$ and $\text{maxocc}(r, a::t) = \text{maxocc}(r_i, a::t)$.

By $\lambda(u)$, we mean the type of node u. For example, in Fig. 6, $\lambda(u_1) = t_1$, $\lambda(u_2) = t_2$, and so on. For an edge $e = (v_1, a, v_2)$ in q_2, we say that e is *answer-reducing* for q_1 if one of the following conditions holds:

(a) $\mu(v_1) \in V(q_1)$ and $\text{minocc}(\delta(\lambda(v_1)), a::\lambda(v_2)) = 0$, i.e., $a::\lambda(v_2)$ is qualified by ? or $*$ in $\delta(\lambda(v_1))$.
(b) $\mu(v_1) \in V(q_1)$, $\text{minocc}(\delta(\lambda(v_1)), a::\lambda(v_2)) \geq 1$, $\text{maxocc}(\delta(\lambda(v_1)), a::\lambda(v_2)) = \infty$, and q_1 already has another edge $(\mu(v_1), a, u_3)$, such that $\lambda(u_3) = \lambda(\mu(v_2))$.
(c) Adding the corresponding edge of e to q_1 yields a new "incoming" edge of $\mu(v_1)$, i.e., $\mu(v_2)$ is a new node and $\mu(v_1) \in V(q_1)$.
(d) Adding the corresponding edge of e to q_1 yields a new weakly biconnected component consisting of three or more edges.
(e) If the corresponding edge of e is added to q_1, then it is under a disjunctive operator of $\delta(\mu(v_1))$ and q_1 has no other edge under the disjunctive operator.

For example, in Fig. 6, (a) applies to (v_3, e, v_2), (c) applies to (v_5, a, v_3), and (d) applies to (v_3, c, v_2). As for (e), assume that

$$\delta(t_1) = a::t_2 \parallel (b::t_3)? \parallel (d::t_2 | f::t_2).$$

If an edge $(\lambda(v_1), d, u)$ is added to q_1, where u is a new node, $\lambda(v_1)$ cannot have an outgoing edge e' labeled by f due to the disjunction operator, meaning that q_1 can no longer match any graph having an edge matching e'. Thus, (v_1, d, v_4) is answer-reducing under the assumption.

We now present our algorithm (Algorithm 3). In lines 1 and 2, biconnected components can be obtained by linear-time depth first search [10]. In line 3, our algorithm finds a set $M(q_1, q_2)$ of node correspondences μ between the nodes of q_1 and q_2. For each correspondence μ in $M(q_1, q_2)$, the algorithm checks if q_1 is contained in q_2 w.r.t. μ, as follows (lines 4–19): if neither q_1 nor q_2 contains any weakly biconnected component of size less than three, we use AddEdge immediately (lines 5 and 6). The AddEdge algorithm adds, for each edge e in q_2, its corresponding edge e' to q_1 if e' is not in q_1 and not answer-reducing (lines 1–6 in AddEdge). Then, the algorithm checks if q_2 is a subgraph of the "extended" q_1, i.e., the query obtained from the original q_1 by adding the non answer-reducing edges (line 7). If this is true, then the algorithm returns true, i.e., $q_1 \subseteq q_2$. Consider the "else" part of the Main algorithm (line 7–15), i.e., the case where q_1 or q_2 contains one or more weakly biconnected components of size three or more. In this case, we first check if q_2 contains a weakly biconnected component c that is not contained in any weakly biconnected component of q_1 (line 10). If so, the result is set to false, since c imposes an extra restriction to q_2 and, thus, q_2 cannot contain q_1. Otherwise, AddEdge is applied to q_1, q_2 (line 13).

Algorithm 3 Main

Input: ShEx schema $S = (\Sigma, \Gamma, \delta)$, queries q_1, q_2
Output: true or false

```
1: BC(q1) ← FindWeaklyBiconnectedComponets(q1)
2: BC(q2) ← FindWeaklyBiconnectedComponets(q2)
3: M(q1, q2) ← FindNodeCorrespondences(q1, q2)
4: for each μ ∈ M(q1, q2) do
5:     if ∀c ∈ BC(q1) |c| < 3 and ∀c ∈ BC(q2) |c| < 3 then
6:         Result ← AddEdge(q1, q2, S, μ)
7:     else
8:         BC3(q1) ← {c ∈ BC(q1) | |c| ≥ 3}
9:         BC3(q2) ← {c ∈ BC(q2) | |c| ≥ 3}
10:        if for some c ∈ BC3(q2), there is no c' ∈ BC3(q1) s.t. c is a subgraph of c' then
11:            Result ← false
12:        else
13:            Result ← AddEdge(q1, q2, S, μ)
14:        end if
15:    end if
16:    if Result = true then
17:        break
18:    end if
19: end for
20: return Result
```

Algorithm 4 AddEdge

Input: ShEx schema $S = (\Sigma, \Gamma, \delta)$, queries q_1, q_2, node correspondence μ
Output: true or false

```
1: for each e ∈ E(q2) do
2:     Let e' be the corresponding edge of e in q1 w.r.t. μ
3:     if e' ∉ E(q1) and e' is not answer-reducing for q1 then
4:         add e' to q1
5:     end if
6: end for
7: if q2 is a subgraph of q1 then
8:     return true
9: end if
10: return false
```

We have the following.

Theorem 1 *Let S be an ShEx schema and q_1, q_2 be queries. If the algorithm returns true, then $q_1 \subseteq q_2$ under S.*

Proof (sketch) Assume that the algorithm returns "true". This means that if condition in line 7 of AddEdge holds, i.e., q_2 is a subgraph of the "extended" q_1. To distinguish the "extended" q_1 and the original one, let q_1' be the query q_1 when the algorithm reaches the line 7 of AddEdge. q_1' may have been added some edges that are not answer-reducing. However, it is easy to see that adding any edges that are not answer-reducing does not change the answer of q_1, i.e., $\mathrm{Ans}(q_1', G) = Ans(q_1, G)$ for any valid graph G. Moreover, since q_2 is a subgraph of q_1', we have $\mathrm{Ans}(q_1', G) \subseteq Ans(q_2, G)$ for any valid graph G. Thus, we have $\mathrm{Ans}(q_1, G) \subseteq Ans(q_2, G)$ for any valid graph G. □

Thus, the soundness of the algorithm holds. The proof of the completeness of the algorithm is a future work, but the experiment shown below suggests that the algorithm can detect query containment without omissions.

Finally, consider the complexity of the problem. Since the query satisfiability problem is subsumed by the complement of the query containment problem and the query satisfiability problem under ShEx is NP-hard [9], the query containment problem can be shown to be intractable; thus, our algorithm requires exponential time in the worst case. However, as shown in the next section, our algorithm runs efficiently for actual RDF data.

Experiments

In this section, we present the result of our experiments. The algorithm was implemented in Python 3.9.0, and all the experiments were executed on a machine with Quad-Core Intel Core i5 CPU, 8.00 GB RAM, and Mac OS Monterey 12.2.1. We assume that node correspondence between query q_1 and q_2 is unknown (except output nodes) for all the experiments.

The metrics for the experiments are based on the following. First, the proposed algorithm is sound as shown in Theorem 1, but its completeness is not given. Therefore, we examine the latter property in the first experiment. Second, as discussed in the previous section, this problem is inherently intractable; therefore, we check how much computation time is required to solve the problem on actual data in the second experiment.

Schemas and Queries

In selecting schema/data, we considered the following points. First, it is desirable to use both synthetic and real data. In addition, since this study targets pattern queries, it is desirable to use a schema including cycles as well as tree structures. Thus, we prepared two ShEx schemas: the one was based on SP2Bench [11], which consists of 11 types. The other was created from a fragment of the Wikidata schema, edition of a written work (E36), which consists of 6 types.

We generated queries based on the following idea: when checking containment of two different queries, the structures of the queries are usually similar to each other, and the structures of the queries rarely differ significantly. Therefore, we first create a "base query", and then, we create "derived queries" from the base query, so that the derived queries have "similar" structures to each other. The derived queries from the same base query constitute the same "query set". Thus, our experiment are conducted as follows:

1. We created ten *base queries*, five for SP2Bench and five for Wikidata schema.
2. From each base query, we created 15 *derived queries*, which constitute a *query set*.

Table 1 List of base queries

Query set name	ShEx schema	Contain biconnected component?	# of edges
SP^{2}Bench_a_0	SP^{2}Bench	Yes	6
SP^{2}Bench_a_1	SP^{2}Bench	Yes	3
SP^{2}Bench_n_2	SP^{2}Bench	No	4
SP^{2}Bench_n_3	SP^{2}Bench	No	5
SP^{2}Bench_n_4	SP^{2}Bench	No	6
Wikidata_a_0	Wikidata	Yes	5
Wikidata_a_1	Wikidata	Yes	5
Wikidata_n_2	Wikidata	No	4
Wikidata_n_3	Wikidata	No	5
Wikidata_n_4	Wikidata	No	6

3. For each query set, we ran the proposed algorithm for ${}_{15}\mathrm{P}_2 = 210$ pairs of 15 queries in the query set and measured execution time, etc.

A base query is constructed as follows: initially, query q is empty. Then, for given schema S (represented by a graph) and query size k (# of edges), we randomly chooses an edge from S and adds it to q, repeating until the size of q reaches k. To do this, we made a program that constructs a base query and we created the $5 \times 2 = 10$ base queries by the algorithm (Table 1).

For each base query, we create 15 derived queries. Let q be a base query, S be an ShEx schema (represented by a graph), and k be a query size. Then, a query of size k derived from q is constructed as follows:

1. Let q_d be an empty query. Add an edge of q to q_d, repeating until the dice coefficient between q and q_d becomes greater than or equal to 0.6.
2. Choose an edge e in S connected to some node in q_d, and add e to q_d, repeating until the size of q_d is equal to k. Return q_d as the result.

For each base query, we specify five sizes $3, 4, \ldots, 7$ and run the above method 3 times for each size; thus, we obtain $5 \times 3 = 15$ queries for each base query. The 15 queries constitute a query set of q.

For example, Fig. 7 shows the base query and a derived query for the SP^{2}Bench_a_0 query set. The base query and the derived query have four edges in common, two edges $(v_2, \text{creator}, v_1)$ and $(v_3, \text{rdf}:, x_1)$ are deleted from the base query, and one edge (v_1, type, v_4) is added to the derived query.

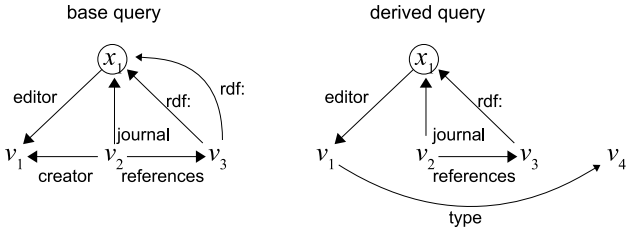

Fig. 7 Base query and derived query for SP^{2}Bench_a_0

Table 2 Precision of the proposed and baseline algorithms

	Proposed algorithm	Baseline
SP^{2}Bench_a_0	1.00	0.976
SP^{2}Bench_a_1	1.00	0.843
SP^{2}Bench_n_2	1.00	0.738
SP^{2}Bench_n_3	1.00	0.857
SP^{2}Bench_n_4	1.00	0.881
Wikidata_a_0	1.00	0.900
Wikidata_a_1	1.00	0.714
Wikidata_n_2	1.00	0.424
Wikidata_n_3	1.00	0.657
Wikidata_n_4	1.00	0.881

Results

Using the two ShEx schemas and the queries obtained above, we conducted two experiments. We first calculated precision and recall, and then, we measured the execution time of the algorithm.

Precision and Recall

We compare the precision and recall of our algorithm and those of a baseline. Here, we used a subgraph isomorphism algorithm (extended for projection) as the baseline, since there is no other containment algorithm for pattern query with projection under ShEx.

Tables 2 and 3 show the precision and recall obtained for the ten query sets, respectively. Overall, our algorithm shows better performance over the baseline. As shown by Theorem 1, if our algorithm returns "true", then we have $q_1 \subseteq q_2$, which is consistent with the precision values of Table 2. Moreover, the recall values in Table 3 show a significant improvement over the baseline. The recall values of the baseline algorithm vary across the different query sets; the more pairs of queries s.t. $q_1 \subseteq q_2$ a query set contains, the lower the recall tends to be.

Table 3 Recall of the proposed and baseline algorithms

	Proposed algorithm	Baseline	# of pairs s.t. $q_1 \subseteq q_2$
SP^{2}Bench_a_0	1.00	0.688	16
SP^{2}Bench_a_1	1.00	0.441	59
SP^{2}Bench_n_4	1.00	0.353	85
SP^{2}Bench_n_5	1.00	0.362	47
SP^{2}Bench_n_6	1.00	0.500	50
Wikidata_a_0	1.00	0.382	34
Wikidata_a_1	1.00	0.178	73
Wikidata_n_4	1.00	0.193	150
Wikidata_n_5	1.00	0.258	97
Wikidata_n_6	1.00	0.419	43

Table 4 Execution time of the proposed and baseline algorithms (s)

	Proposed algorithm	Baseline
SP^{2}Bench_a_0	0.000468	0.130
SP^{2}Bench_a_1	0.000874	0.124
SP^{2}Bench_n_2	0.000661	3.48
SP^{2}Bench_n_3	0.000716	3.50
SP^{2}Bench_n_4	0.000691	3.75
Wikidata_a_0	0.000366	0.147
Wikidata_a_1	0.000378	0.128
Wikidata_n_2	0.000965	3.75
Wikidata_n_3	0.000502	3.53
Wikidata_n_4	0.000577	3.83

Execution Time

Then, we measured the execution time of our algorithm. The algorithm uses ShEx types to narrow the search space when finding node correspondences. To see the effect of this, we also prepared a baseline algorithm, which is identical to the proposed algorithm except that the baseline does not use ShEx types when finding node correspondence. Execution time was measured using Ruby's Benchmark module.

For each pair of queries, we executed each algorithm ten times and calculated its mean execution time. Table 4 shows the mean execution time of 210 pairs for each query set. Table 5 shows the mean execution time for each query size under the SP^{2}Bench schema. Similarly, Table 6 shows the mean execution time for each query size under the Wikidata's ShEx schema.

From the results of Table 4, we can see that the proposed algorithm runs much faster than the baseline. This indicates that ShEx types can effectively reduce the search space for finding node correspondences between queries.

As shown in Tables 5 and 6, the execution time tends to increase as the query size increases. However, the increase is proportional to the query size, and it is unlikely that the execution time increases considerably even if the query size increases slightly.

Table 5 Breakdown of execution time for SP^{2}Bench schema (s)

$q_1 \backslash q_2$	3	4	5	6	7
3	0.000354	0.000362	0.000391	0.000438	0.000536
4	0.000424	0.000476	0.000524	0.000638	0.000795
5	0.000398	0.000484	0.000614	0.000701	0.000943
6	0.000469	0.000681	0.000813	0.000793	0.001146
7	0.000691	0.000799	0.001091	0.001230	0.001291

Table 6 Breakdown of execution time for Wikidata schema (s)

$q_1 \backslash q_2$	3	4	5	6	7
3	0.000286	0.000317	0.000342	0.000359	0.000384
4	0.000340	0.000416	0.000427	0.000446	0.000536
5	0.000415	0.000426	0.000466	0.000507	0.000621
6	0.000415	0.000477	0.000572	0.000588	0.000759
7	0.000591	0.000980	0.001057	0.001135	0.001086

Related Work

Query containment has been a popular problem in data management field including relational database and XML. For example, Shmueli considered complexity of the containment problem for datalog queries [12], Calvance et al. considered the decidability of query containment under schema [13], and Wood proposed an algorithm for solving XPath query containment problem under DTDs [14]. Studies on XPath query containment with/without schemas are summarized in [15]. Chen and Rundensteiner study the problem of XQuery containment [16].

As for RDF/graph data, Pichler and Skritek studied query containment for a SPARQL fragment without schema [17]. Abbas et al. studied complexity of SPARQL containment under ShEx without projection [18]. Saleem et al. proposed a framework of SPARQL query containment without schema [19]. Chekol et al. studied complexity of query containment problem for SPARQL fragments under RDF Schema [20]. Mailis et al. proposed an index for RDF query containment without schema [21]. Mirko et al. propose an SPQRQL query containment solver without ShEx [22].

To the best of our knowledge, however, no studies on pattern graph with projection containment under ShEx have been done. Moreover, most previous studies for query containment are based on logic, and our study is unprecedented in adopting a graph-theoretic approach. Thus, we believe

that our approach gives a novel different perspective on solving the problem.

A problem related to query containment is query satisfiability, which is to decide, for given query q and a schema S, whether there is a datum valid for S, such that the answer of q is nonempty. For the problem, a number of studies have been made so far. For example, Benedikt et al. considered the XPath satisfiability problem under DTDs [23], and Geneves and Layaida proposed a system for checking satisfiability of XPath queries [24]. Ishihara et al. proposed a subclass of DTDs under which XPath satisfiability problem can be solved efficiently [25]. Zhang et al. considered satisfiability of SPARQL patterns without schema [26]. Matsuoka proposed an algorithm for checking satisfiability of pattern queries under ShEx [9].

Conclusion

In this paper, we proposed an algorithm for checking containment of pattern queries under ShEx schema. Our algorithm uses ShEx schema to reduce the search space of finding a correspondence between nodes of queries. Then, the algorithm extends q_1 by adding edges that are not answer-reducing, and check if q_2 is contained in q_1 as a subgraph. Moreover, our algorithm is shown to be sound.

In our experiments, we verified that the results of our algorithm are correct for all query pairs generated in the experiments. The results of another experiment suggest that types of nodes obtained from ShEx schema can reduce the search space for finding corresponding nodes between queries. In addition, we showed that the weakly biconnected component and the size of the queries are the main factors in the efficiency of the algorithm.

However, our algorithm has several limitations that need to be addressed. First, the model of the schema used in this study is based on the core part of ShEx and does not support full of the constraints, e.g., Inverse Triple Constraints and Negative Triple Constraints. It would be interesting to consider how such constraints can be handled by extending our model and algorithm. In addition, our algorithm is shown to be sound in Theorem 1, but the converse is only verified by experiments. Therefore, giving a proof of this is a highly interesting challenge. In particular, it is interesting to consider whether the converse holds without any restrictions on ShEx, and if not, what restrictions should be placed. Finally, ShEx has two typing semantics: single-type semantics and multi-type semantics [2]. While this paper assumes the former semantics, adapting our algorithm to the latter semantics is an interesting challenge.

Acknowledgements The authors express sincere thanks to anonymous reviewers for their invaluable and insightful comments that greatly help us to improve this paper. This work was partly supported by JSPS KAKENHI under Grant No. 21K11900.

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

References

1. Fujimoto H, Suzuki N. A simple algorithm for checking pattern query containment under shape expression schema. In: Proceedings of the 18th international conference on web and information systems and technologies (WEBIST 2022). 2022;278–85.
2. Staworko S, Boneva I, Gayo JEL, Hym S, Prud'hommeaux EG, Solbrig HR. Complexity and expressiveness of ShEx for RDF. In: Proceedings of 18th international conference on database theory (ICDT 2015). 2015;195–211.
3. Thornton K, Solbrig H, Stupp GS, Labra Gayo JE, Mietchen D, Prud'hommeaux E, Waagmeester A. Using shape expressions (ShEx) to share RDF data models and to guide curation with rigorous validation. In: Proceedings of the European semantic web conference (ESWC 2019). 2019;606–20.
4. Chhabra S, Aiden MK, Sabharwal SM, Al-Asadi M. In: Ahad MA, Casalino G, Bhushan B, editors. 5G and 6G technologies for smart city. Cham: Springer; 2023. p. 335–65.
5. Yadav L, Mitra M, Kumar A, Bhushan B, Al-Asadi MA. In: Sharma DK, Sharma R, Jeon G, Polkowski Z, editors. Nullifying the prevalent threats in IoT based applications and smart cities using blockchain technology. Singapore: Springer; 2023. p. 241–61.
6. Nesi P, Bellini P. Assessing RDF graph databases for smart city services. In: Proceedings of the 23rd international DMS conference on visual languages and sentient systems. 2017.
7. Bellini P, Nesi P. Performance assessment of RDF graph databases for smart city services. J Vis Lang Comput. 2018;45:25.
8. Gayo JEL, Prud'hommeaux E, Boneva I, Kontokostas D. Validating RDF data. Cham: Springer; 2018.
9. Matsuoka S, Suzuki N. Detecting unsatisfiable pattern queries under shape expression schema. In: Proceedings of the 16th international conference on web and information systems and technologies. 2020;285–91.
10. Hopcroft J, Tarjan R. Algorithm 447: efficient algorithms for graph manipulation. Commun ACM. 1973;16(6):372–8.
11. Schmidt M, Hornung T, Lausen G, Pinkel C. SP2Bench: a SPARQL performance benchmark. In: Proceedings of the 25th international conference on data engineering (ICDE 2009). 2009;371–93.
12. Shmueli O. Decidability and expressiveness aspects of logic queries. In: Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems (PODS'87). 1987;237–49.
13. Calvanese,D, Giacomo GD, Lenzerini M. On the decidability of query containment under constraints. In: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems (PODS'98). 1998.
14. Wood PT. Containment for XPath fragments under DTD constraints. In: Proceedings of the 9th international conference on database theory (ICDT'03). 2003;300–14.
15. Schwentick T. XPath query containment. SIGMOD Rec. 2004;33(1):101–9.
16. Chen L, Rundensteiner EA. XQuery containment in presence of variable binding dependencies. In: Proceedings of the

14th international conference on world wide web. WWW '05. 2005;288–97.
17. Pichler R, Skritek S. Containment and equivalence of well-designed SPARQL. In: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems. 2014;39–50.
18. Abbas A, Genevés P, Roisin C, Layaïda N. SPARQL query containment with ShEx constraints. In: Proceedings of advances in databases and information systems (ADBIS 2017). 2017;343–56.
19. Saleem M, Stadler C, Mehmood Q, Lehmann J, Ngomo A-CN. SQC Framework: SPARQL query containment benchmark generation framework. In: Proceedings of the knowledge capture conference. K-CAP 2017. 2017.
20. Chekol MW, Euzenat J, Genevès P, Layaïda N. SPARQL query containment under schema. J Data Semant. 2018;7(3):133–54.
21. Mailis T, Kotidis Y, Nikolopoulos V, Kharlamov E, Horrocks I, Ioannidis Y. An efficient index for RDF query containment. In: Proceedings of the 2019 international conference on management of data. 2019;1499–16.
22. Spasić M, Janičić MV. SpeCS-SPARQL query containment solver. In: Proceedings of the 2020 zooming innovation in consumer technologies conference (ZINC). 2020;31–5.
23. Benedikt M, Fan W, Geerts F. XPath satisfiability in the presence of DTDs. J ACM. 2008;55:2.
24. Genevès P, Layaïda N. A system for the static analysis of XPath. ACM Trans Inf Syst. 2006;24(4):475–502.
25. Ishihara Y, Suzuki N, Hashimoto K, Shimizu S, Fujiwara T. XPath satisfiability with parent axes or qualifiers is tractable under many of real-world DTDs. In: Proceedings of the 14th international symposium on database programming languages (DBPL 2013). 2013.
26. Zhang X, den Bussche JV, Picalausa F. On the satisfiability problem for SPARQL patterns. J Artif Intell Res. 2016;55:403–28.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.