A Simple Algorithm for Checking Pattern Query Containment under Shape Expression Schema

Haruna Fujimoto and Nobutaka Suzuki

University of Tsukuba, 1-2 Kasuga, Tsukuba, 305-8550, Japan

Keywords: RDF, Query Containment, Graph Data.

Abstract: Query containment is one of the major fundamental problems for various kinds of data including RDF/graph, and related to many important practical problems, e.g., determining independence of queries from updates and rewriting queries using views. In this paper, we consider a query containment problem under Shape Expression (ShEx), where query is defined as pattern graph with projection. We adopt a graph-theoretic approach to cope with the containment problem, and propose a simple sound algorithm for solving the problem. In our preliminary experiments, we first verified that the results of our algorithm are correct for all pairs of queries generated in the experiments. We also show that types of ShEx schema can be used to reduce the search space for checking pattern query containment.

1 INTRODUCTION

For over years, RDF/graph data has been used in a wide variety of fields. For various kinds of data including RDF/graph, query containment is one of the major fundamental problems. Query containment is a 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 practical problems, e.g., 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 being considered by Shape Expression Community Group. ShEx is designed for capturing structural features of RDF/graph data. A ShEx schema assigns types to the nodes of an RDF/graph data and allows to define a set of types that impose structural constraints on nodes and their immediate neighborhood with regular bag expression (RBE) (Staworko et al., 2015). ShEx is useful in multiple contexts, e.g., model development, regacy review, documentation of models and already used in a variety of areas (Thornton et al., 2019).

ShEx shares many fundamental features with Shapes Constraint Language (SHACL) (Gayo et al., 2018), thus the result of this paper can also be applied to SHACL as well. As for query language, we focus on pattern graph with projection. For example, Fig. 1 depicts a tiny example consisting of three nodes but only the value of circled node $u_1$ is output. Intuitively, this query outputs any student taking a course taught by his/her supervisor.

Figure 1: Example of pattern query with projection.

We adopt a simple graph-theoretic approach to cope with the containment problem. 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 $q_1 \subseteq q_2$ if the projections and $S$ are ignored, $q_1 \nsubseteq q_2$ otherwise.

To cope with this problem, we devised a novel simple algorithm for checking pattern query containment under ShEx schema. For given pattern queries
Figure 2: Queries q1 and q2.

$q_1$ and $q_2$, the algorithm firstly 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 by using the types of ShEx schema 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 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 $q_2$. The algorithm is shown to be sound but the proof of its completeness is still ongoing. In our preliminary experiments, we verified that the results of our algorithm are correct for all pairs of queries generated in the experiments. We also showed that types of ShEx schema can be used to reduce the search space for checking pattern query containment.

1.1 Related Work

Query containment has been a popular problem in data management field including relational database and XML (e.g. (Wood, 2003)). As for RDF/graph data, Pichler and Skritek studied query containment for a SPARQL fragment without schema (Pichler and Skritek, 2014). Abbas et al. studied complexity of SPARQL containment under ShEx without projection (Abbas et al., 2017). Saleem et al. proposed a framework of SPARQL query containment without schema (Saleem et al., 2017). Chekol et al. studied complexity of query containment problem for SPARQL fragments under RDF Schema (Chekol et al., 2018). Saleem et al. proposed an index for RDF query containment without schema (Saleem et al., 2019). To the best of our knowledge, however, no studies on pattern graph with projection containment under ShEx has been made.

2 PRELIMINARIES

Let $\Sigma$ be a set of labels. A labeled directed graph (graph for short) over $\Sigma$ 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 $V(q) = \{u_1,u_2,u_3\}$, $E(q) = \{((u_1,\text{supervisor},u_3),(u_1,\text{takes},u_2),(u_3,\text{teaches},u_2))\}$, and $P = \{u_1\}$. By $Ans(q,G)$, 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 $Ans(q,G) = \{(v_1),(v_2)\}$.

The content model of type in ShEx can be modeled as regular bag expression (RBE) (Staworko et al., 2015). RBE is defined similarly to regular expression except that RBE uses unordered concatenation instead of ordered concatenation. Let $\Gamma$ be a set of types. Then RBE over $\Sigma \times \Gamma$ is recursively defined as follows.

- $\varepsilon$ and $a::t \in \Sigma \times \Gamma$ are RBEs. $\varepsilon$ 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 $r? = \varepsilon|\varepsilon r$.

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 $a::t_1\parallel c::t_3$ and $b::t_2\parallel c::t_3$ but also $c::t_3\parallel a::t_1$ and $c::t_3\parallel b::t_2$. 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 $\Gamma$ is a set of types and $\delta$ is a function from $\Gamma$ to the set of RBEs over $\Sigma \times \Gamma$. For example, let $S = (\Sigma,\Gamma,\delta)$ be a ShEx schema, where $\Sigma =$
\{\text{takes, supervisor, teaches}\}, \Gamma = \{t_1, t_2, t_3\}, and
\[\delta(t_1) = \text{(takes :: t_2)}^* \parallel \text{(supervisor :: t_3)}?,\]
\[\delta(t_2) = \varepsilon,\]
\[\delta(t_3) = \text{(teaches :: t_2)}^*\].

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

3 ALGORITHM

Our algorithm is essentially based on the node correspondence between \(q_1\) and \(q_2\). Such a correspondence may already be known in some cases, e.g., comparing an updated query and its original one. But this is not always the case. Thus, our algorithm firstly finds a node correspondence between \(q_1\) and \(q_2\) (Sec. 3.1) if necessary, and then under the obtained node correspondence, the algorithm checks the containment of \(q_1\) and \(q_2\) (Sec. 3.2).

3.1 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 a correspondence of between their non-output nodes. This is done by the following steps.

1. Let \(S\) be a ShEx schema. By using \(S\), we identify the type(s) of each node in \(q_1\) and \(q_2\). This is done by an extension of an algorithm for checking satisfiability of pattern queries (Matsuoka and Suzuki, 2020) (details are omitted because of space limitations).

2. By comparing the type(s) of each node obtained in step (1), we find correspondence(s) between the nodes of \(q_1\) and \(q_2\). Their correspondences are found from the output node of \(q_1\) in the order of connection. Two nodes do not correspond to each other even if they are of the same type, when there is no correspondence between their adjacent nodes. For example, \(u_2\) of \(q_1\) corresponds to \(v_1\) of \(q_2\) in Fig 4. This is because \(x_1\) an adjacent node of \(u_2\) corresponds to \(y_1\), an adjacent node of \(v_1\). On the other hand, \(u_2\) of \(q_1\) cannot correspond to \(v_2\) of \(q_2\) in Fig 4 because there is no correspondence between their adjacent nodes. For each obtained correspondence, we compute a maximum common edge subgraph of \(q_1\) and \(q_2\) under the correspondence. The problem is NP-hard, but the types of nodes obtain in step (1) can reduce the search space of the problem.

3. Among the correspondences obtained in step (2), output the correspondence that yields the maximum common subgraph of \(q_1\) and \(q_2\).

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 \(\mu(u) = v_{nil}\) if there is no node corresponding to \(u\), where \(v_{nil}\) is a new node not in \(V(q_2)\). For an edge \(e = (v, a, v')\) in \(q_2\), \((\mu(v), a, \mu(v'))\) is called the corresponding edge of \(e\) in \(q_1\).

We explain the above steps (1) to (3) by an example. Consider queries \(q_1\) and \(q_2\) shown in Fig. 4, and suppose that in step (1) the type of each node is obtained as shown in the figure. Consider step (2). By the assumption, \(x_1\) 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. Since \(u_2\) of type \(t_2\), \(u_2\) may correspond to \(v_1\) and \(v_2\). However, by checking edges adjacent to \(u_2\) it is impossible for \(u_2\) to correspond to \(v_2\), and we know that \(u_2\) is able to correspond to only \(v_1\). Since \(u_1\) is of type \(t_3\) but there is no node of type \(t_3\) except \(y_2\), we know that \(u_2\) is not corresponding to \(u_1\). Similarly, there is no node corresponding to \(v_2\). Therefore, we have \(\mu(x_1) = y_1, \mu(x_2) = y_2, \mu(u_2) = v_1\), and \(\mu(u_1) = v_{nil}\) (and \(\mu(v_2) = u_{nil}\)). Based on this correspondence, we create adjacency matrices of \(q_1\) and \(q_2\) (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.

In this case we have only one correspondence, but in general there may be more than one correspondence between two queries. In such a case, for each possible correspondence with each node associated with one type, we compute the size of the common edge subgraph under the correspondence, and choose the maximum one among them.

3.2 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, the resulting undirected subgraph remains connected. A subgraph $G'$ of $G$ is weakly biconnected component if $G'$ is a maximal weakly biconnected subgraph.

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.”

Next, we will explain the idea of “answer-reducing” edge. 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, and let $S = (\Sigma, \Gamma, \delta)$ be a ShEx schema, where $\Gamma = \{t_1, t_2, t_3\}$ and $\delta$ as defined as follows:

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

$$
\delta(t_2) = e,
$$

$$
\delta(t_3) = c :: t_2 \parallel (e :: t_2)?
$$

Then $(v_1, d, v_4)$ is not answer-reducing, since 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, since 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, $(v_3, e, v_5)$ is answer-reducing, since in $\delta(t_5) e :: t_2$ is qualified by $?$, meaning that every node of type $t_5$ 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 $a :: t \subseteq \Sigma :: \Gamma$. The minimum occurrence and maximum occurrence of $a :: t$, denoted $\minocc(r, a :: t)$ and $\maxocc(r, a :: t)$, respectively, are defined as follows.

- If $r = a :: t$, then $\minocc(r, a :: t) = \maxocc(r, a :: t) = 1$.
- If $r = r^+$ and $a :: t$ is in $r'$, then $\minocc(r, a :: t) = \maxocc(r, a :: t) = 0$ and $\maxocc(r, a :: t) = \infty$.
- If $r = r^+$ and $a :: t$ is in $r'$, then $\minocc(r, a :: t) = \minocc(r', a :: t)$ and $\maxocc(r, a :: t) = \maxocc(r', a :: t)$.
- If $r = r^+$ and $a :: t$ is in $r'$, then $\minocc(r, a :: t) = \minocc(r', a :: t) = 0$ and $\maxocc(r, a :: t) = \maxocc(r', a :: t)$.

By $\lambda(u)$ we mean the type of node $u$. For example, in Fig. 4 $\lambda(x_1) = t_1, \lambda(u_2) = u_3$, 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 $\minocc(\delta(\lambda(v_1)), a :: \lambda(v_2)) = 0$, i.e., $a :: \lambda(v_2)$ is qualified by $?$ or $*$ in $\delta(\lambda(v_1))$. 

281
(b) \( \mu(v_1) \in V(q_1), \text{minocc}(\delta(\lambda(v_1)), a :: \lambda(v_2)) \geq 1, \text{maxocc}(\delta(\lambda(v_1)), a :: \lambda(v_2)) = \infty, \text{and} \ q_1 \text{ already has another edge } (\mu(v_1), a, u_3) \text{ such that } \lambda(u_3) = \lambda(\mu(v_2)). \)

(c) The corresponding edge of \( e \) is a new "incoming" one, 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.

(e) The corresponding edge of \( e \) 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 \( (v_1, e, v_2) \), (c) applies to \( (v_5, a, v_3) \), and (d) applies to \( (v_3, c, v_2) \).

Algorithm 1: Main.

**Input:** ShEx schema \( S = (\Sigma, \Gamma, \delta) \), queries \( q_1, q_2 \)

**Output:** true or false

1. \( C(q_1) \leftarrow \text{FindWeaklyBiconnectedComponents}(q_1) \)
2. \( C(q_2) \leftarrow \text{FindWeaklyBiconnectedComponents}(q_2) \)
3. \( X(q_1, q_2) \leftarrow \text{FindNodeCorrespondence}(q_1, q_2) \)
4. \( \text{for each } x \in X(q_1, q_2) \) do
5. \( \quad \text{if } \forall c \in C(q_1) \ |c| < 3 \text{ and } \forall c \in C(q_2) \ |c| < 3 \) then
6. \( \quad \quad \text{Result} \leftarrow \text{AddEdge}(q_1, q_2, S, x) \)
7. \( \quad \text{else} \)
8. \( \quad \quad M(q_1) \leftarrow \{ c \in C(q_1) \ | \ |c| \geq 3 \} \)
9. \( \quad \quad M(q_2) \leftarrow \{ c \in C(q_2) \ | \ |c| \geq 3 \} \)
10. \( \quad \text{Result} \leftarrow \text{IsInclude}(M(q_1), M(q_2), q_1, q_2, S, x) \)
11. \( \quad \text{if Result} = \text{true} \) then
12. \( \quad \quad \text{break} \)
13. \( \quad \text{return Result} \)

Algorithm 2: AddEdge.

**Input:** ShEx schema \( S = (\Sigma, \Gamma, \delta) \), queries \( q_1, q_2 \), correspondence \( x \) between the nodes of \( q_1 \) and \( q_2 \)

**Output:** true or false

1. \( \text{for each } e \in E(q_2) \) do
2. \( \quad \text{Let } e' \text{ be the corresponding edge of } e \text{ in } q_1 \text{ under } x \)
3. \( \quad \text{if } e' \notin E(q_1) \text{ and } e' \text{ is not answer-reducing for } q_1 \) then
4. \( \quad \quad \text{add } e' \text{ to } q_1 \)
5. \( \quad \text{if } q_2 \text{ is a subgraph of } q_1 \) then
6. \( \quad \quad \text{return true} \)
7. \( \quad \text{return false} \)

We now present our algorithm (Algorithm 1). In lines 1 and 2, biconnected components can be obtained by linear-time depth first search (Hopcroft and Tarjan, 1973). In line 3, our algorithm finds a set \( X(q_1, q_2) \) of node correspondences between \( q_1 \) and \( q_2 \). For each correspondence \( x \) in \( X(q_1, q_2) \), the algorithm checks if \( q_1 \) is contained in \( q_2 \) under \( x \), as follows (lines 4 to 12). 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). This 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, and then check if \( q_2 \) is a subgraph of the extended \( q_1 \). If this is true, then the algorithm returns true, i.e., \( q_1 \subseteq q_2 \). If \( q_1 \) or \( q_2 \) contains one or more weakly biconnected components of size three or more, we use IsInclude (lines 8 to 10). This checks if \( q_2 \) contains a weakly biconnected component \( c \) that is not contained in any weakly biconnected component of \( q_1 \) (line 1 of IsInclude). If so, the algorithm returns 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 3). We have the following.

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

We are considering the completeness of the algorithm. We expect that the completeness also holds at least under certain restricted ShEx schema.

## 4 PRELIMINARY EXPERIMENTS

We present the result of our preliminary 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, 8.00GB RAM, and Mac OS Monterey 12.2.1.

We made two ShEx schemas for RDF data generated by SP2Bench (Schmidt et al., 2009) (consisting of 11 types) and a fragment of Wikidata schema (consisting of 6 types). Queries were created as follows. The number of edges in each query was between 3 and 7, and for each size (3, 4, \ldots, 7) three pattern graphs were created, where one contained more than two biconnected components and the others not. Thus we obtained \( 5 \times 3 = 15 \) queries for each of the
ShEx schemas. We examined all permutations of \( q_1 \) and \( q_2 \) from the 15 queries, i.e., we ran the proposed algorithm for \( 15 \times 2 = 30 \) pairs of queries. For every pair, we assumed that the correspondence of non-output nodes is unknown.

First, we verified the results of our algorithm, and found that all the results of our algorithm for the 210 pairs were correct. This suggests that in most cases our algorithm can solve the containment problem correctly, although only the soundness of our algorithm has been shown by Theorem 1.

Second, we compare our algorithm and a baseline algorithm. Here, the baseline algorithm finds the correspondence of nodes without using schema types, and the rest part is identical to that of our algorithm. Thus, this experiment is to measure the effect of ShEx types on the efficiency of our algorithm.

Table 1 shows the average execution time for the 210 pairs for each ShEx schema. As shown in the table, our algorithm is much faster than the baseline. The SP2Bench result shows that the execution time is reduced to about 1/520 by using ShEx types, while the Wikidata result shows that the execution time is reduced to about 1/385. These results show that the types of nodes obtained by ShEx schema can reduce the search space of finding node correspondences between queries.

Third, we investigated the execution time of our algorithm further. Tables 2 and 3 show breakdowns of the average execution time by the sizes of \( q_1 \) (row) and \( q_2 \) (column). As shown in the tables, the algorithm can check the containment of pattern queries under ShEx schema in relatively short time, and the execution time tends to increase with the size of query.

Tables 4 and 5 show other breakdowns by the number of weakly biconnected components contained in the queries, for the following three cases: (a) both \( q_1 \) and \( q_2 \) contain more than two weakly biconnected components, (b) only \( q_1 \) contains more than two weakly biconnected components, (c) only \( q_2 \) contains more than two weakly biconnected components, and (d) neither \( q_1 \) nor \( q_2 \) contains more than two weakly biconnected components. Interestingly, these suggest that the execution time tends to be smaller when query \( q_2 \) contains weakly biconnected components with more than two nodes. A possible reason is that the algorithm can sometimes avoid executing AddEdge if given pattern query contains weakly biconnected components with more than two nodes, i.e., when the if test in line 1 of IsInclude holds, the execution of AddEdge in line 3 is avoided.

Figures 7 and 8 plot the number of correspondences, i.e., the size of \( X(q_1, q_2) \) (y-axis) and the execution time of the algorithm (x-axis) for each pair of queries. As shown in the tables, as the size of \( X(q_1, q_2) \) becomes larger, the execution time increases accordingly. This suggests that reducing the size of \( X(q_1, q_2) \) is important to solve the problem more efficiently.

5 CONCLUSION

In this paper, we proposed an algorithm for checking containment of pattern queries under ShEx schema.

Our algorithm uses the ShEx schema to reduce the search space of finding a correspondence between nodes of queries. Then, the algorithm extends the pattern graph using the ShEx schema. This allows us to find containment that cannot be found by existing
Table 2: Breakdown of average execution time by query size (SP2Bench).

<table>
<thead>
<tr>
<th>Query Size</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (sec)</td>
<td>2.24 × 10⁻⁴</td>
<td>2.49 × 10⁻⁴</td>
<td>3.20 × 10⁻⁴</td>
<td>8.83 × 10⁻⁴</td>
<td>4.77 × 10⁻⁴</td>
</tr>
</tbody>
</table>

Table 3: Breakdown of average execution time by query size (Wikidata).

<table>
<thead>
<tr>
<th>Query Size</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (sec)</td>
<td>1.38 × 10⁻⁴</td>
<td>2.73 × 10⁻⁴</td>
<td>2.06 × 10⁻⁴</td>
<td>5.44 × 10⁻⁴</td>
<td>3.83 × 10⁻⁴</td>
</tr>
</tbody>
</table>

Table 4: Breakdown of average execution time by the number of weakly biconnected components (SP2Bench).

<table>
<thead>
<tr>
<th># of pairs</th>
<th>average execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>20</td>
</tr>
<tr>
<td>(b)</td>
<td>50</td>
</tr>
<tr>
<td>(C)</td>
<td>50</td>
</tr>
<tr>
<td>(d)</td>
<td>90</td>
</tr>
<tr>
<td>total</td>
<td>210</td>
</tr>
</tbody>
</table>

Table 5: Breakdown of average execution time by the number of weakly biconnected components (Wikidata).

<table>
<thead>
<tr>
<th># of pairs</th>
<th>average execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>20</td>
</tr>
<tr>
<td>(b)</td>
<td>50</td>
</tr>
<tr>
<td>(C)</td>
<td>50</td>
</tr>
<tr>
<td>(d)</td>
<td>90</td>
</tr>
<tr>
<td>total</td>
<td>210</td>
</tr>
</tbody>
</table>

ShEx has more functions not discussed in this paper (e.g., negation). Thus we need to consider extending our algorithm to adopt such functions.

**ACKNOWLEDGMENTS**

This work was partly supported by JSPS KAKENHI Grant Number 21K11900.

**REFERENCES**


