Detecting Unsatisfiable Pattern Queries under Shape Expression Schema

Shiori Matsuoka[1] and Nobutaka Suzuki[2]

[1]Graduate School of Library, Information and Media Studies, University of Tsukuba, 1-2, Kasuga, Tsukuba, Ibaraki, Japan

[2]Faculty of Library, Information and Media Science, University of Tsukuba, 1-2, Kasuga, Tsukuba, Ibaraki, Japan

Keywords: Shape Expression, Pattern Query, Satisfiability.

Abstract: Among queries for RDF/graph data, pattern query is the most popular and important one. A pattern query that returns empty answer for every valid graph is clearly useless, and such a query is called *unsatisfiable*. Formally, we say that a pattern query q is unsatisfiable under a schema S if there is no valid graph g of S such that the result of q over g is nonempty. It is desirable that unsatisfiable pattern queries can be detected efficiently before being executed since unsatisfiable query may require much execution time but always reports empty answer. In this paper, we focus on Shape Expression (ShEx) as schema, and we propose an algorithm for detecting unsatisfiable pattern queries under a given ShEx schema. Experimental results suggest that our algorithm can determine the satisfiability of pattern query efficiently.

1 INTRODUCTION

Over many years, RDF/graph has been a popular data model and used for various kinds of applications, Linked Open Data, social networks, citation graph, and so on. For such data, RDF Schema (RDFS) is sometimes used as a schema definition language. However, RDFS is an ontology language rather than a schema language and is not necessarily suitable for describing structures of graph data (Staworko et al., 2015). Due to this, a new schema language called Shape Expression (ShEx) has been considered under W3C Draft Community Group (Baker and Prud'hommeaux, 2019). ShEx is designed for capturing structural features of RDF data rather than its ontological semantics, and already used in a variety of areas (Thornton et al., 2019).

Among queries for RDF/graph data, pattern query is the most popular and important one. A pattern query that returns empty answer for every valid graph is clearly useless, and such a query is called *unsatisfiable*. Formally, for a pattern query q and a ShEx schema S, we say that q is *unsatisfiable* under S if there is no valid graph G of S such that the answer of q for G is not empty. As a simple example, consider the ShEx schema in Fig. 1 having the definitions of four types t_1, t_2, t_3, t_4. Then Fig. 2 illustrates an example of pattern query over the schema. Since v_1 cannot have "tel" and "email" at the same time due to the definition of t_2, the pattern query is unsatisfiable. Here, suppose that a user writes an unsatisfiable query q and that he/she executes q over a graph data G. Then q traverses nodes/edges of G but fails to find any answer. Since the size of recent RDF/graph data tend to be very large, executing such a query may requires huge computation cost. Therefore, it is desirable that unsatisfiable queries can be detected efficiently before being executed.

In this paper, we propose an algorithm for detecting unsatisfiable pattern queries under ShEx schema. For a pattern query q and a ShEx schema S, the algorithm decomposes S into connections between types of S, then checks the matchability between the type connections and edges of q. If every edge of q is "safely" matched by a connection in S, then the algorithm answers that "q is satisfiable," otherwise "q is unsatisfiable." Although this problem is intractable in general, our preliminary experiments suggest that our algorithms can detect unsatisfiable pattern queries efficiently.

Recently, an architectural schema for business document processing is proposed (Cristani et al., 2018). Business documents are large amounts of date and may be processed using ShEx. For example, some efforts have been spent upon such documents, in which unifying semantics by ShEx could be fruitfully used. Our problem is worth considering in that field.

Matsuoka, S. and Suzuki, N.
Detecting Unsatisfiable Pattern Queries under Shape Expression Schema.
DOI: 10.5220/0010171602850291
In *Proceedings of the 16th International Conference on Web Information Systems and Technologies (WEBIST 2020)*, pages 285-291
ISBN: 978-989-758-478-7
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

```
< t1 >{
   student@ < t2 >*
}
< t2 >{
   supervisor@ < t3 >? ||
   takes@ < t4 >+ ||
   (tel xsd : string | email xsd : string)
}
< t3 >{
   teaches@ < t4 >+
}
< t4 > {
}
```

Figure 1: Example of ShEx schema.

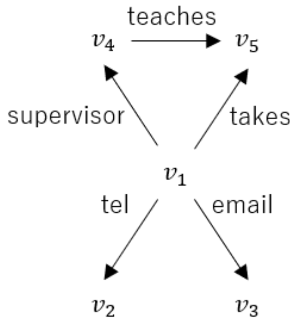

Figure 2: Example of unsatisfiable pattern query.

1.1 Related Work

Satisfiablity of query has been a major problem of database and document management field. A number of studies on XPath satisfiability problem under DTDs or XML Schema have been made, e.g., (Groppe and Groppe, 2007; Montazerian et al., 2007; Benedikt et al., 2008; Ishihara et al., 2013; Figueira, 2018). Geneves et al. proposes a comprehensive tool for checking satisfiability of XPath expressions under schema (Geneves et al., 2011). However, XML is based on ordered tree data model and XPath query can also be represented by tree, and thus their query/data models are quite different from that of this paper. Zhang et al. consider satisfiability of pattern query without schema (Zhang et al., 2016). To the best of the authors' knowledge, however, no studies on satisfiability of pattern queries under ShEx schemas have been made so far.

2 PRELIMINARIES

Let Σ be a set of labels. A *labeled directed graph* (*graph* for short) is denoted $G = (V, E)$, where V is a set of nodes and and $E \subseteq V \times \Sigma \times V$ is a set of *edges*. Let $e \in E$ be an edge labeled by $l \in \Sigma$ from a node $v \in V$ to a node $v' \in V$. Then e is denoted (v, l, v'), v is called *source*, and v' is called *target*. A *pattern query* (*query* for short) is also represented as a graph $q = (V(q), E(q))$. For example, Fig. 3 illustrates a graph G and Fig. 4 illstrates a query q and its answer over G.

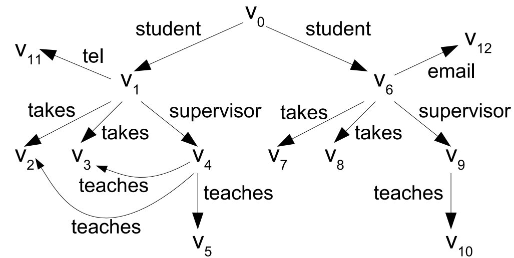

Figure 3: Example of valid graph G.

In XML data model, sibling nodes are ordered. On the other hand, in RDF/graph data model the order among sibling nodes are less significant and thus sibling nodes are treated as unordered. Thus ShEx uses regular bag expression (RBE) to represent the node type (Staworko et al., 2015). RBE is defined similar to regular expressions 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.
- If $r_1, r_2, \cdots, r_k$ are RBEs, then $r_1 | r_2 | \cdots | r_k$ is an RBE, where $|$ denotes disjunction.
- If $r_1, r_2, \cdots, r_k$ 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 $r^{[n,m]}$ is an RBE, where $n \leq m$. $[n, m]$ is called *interval*. In particular, $r^? = r^{[0,1]}$, $r^* = r^{[0,\infty]}$, and $r^+ = r^{[1,\infty]}$.

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 $c :: t_3\ b :: t_2$.

A *ShEx schema* is denoted $S = (\Sigma, \Gamma, \delta)$, where Γ is a set of *types*, δ is a function from Γ to the set of RBEs over $\Sigma \times \Gamma$. For example, the schema in Fig. 1 can be denoted $S = (\Sigma, \Gamma, \delta)$, where

$$
\begin{aligned}
\Sigma &= \{\text{teaches}, \text{student}, \text{supervisor}, \text{takes}, \text{tel}, \\
&\quad\ \text{email}\}, \\
\Gamma &= \{t_1, t_2, t_3, t_4, t_5\}, \\
\delta(t_1) &= (\text{student} :: t_2)^*, \\
\delta(t_2) &= (\text{supervisor} :: t_3)^? \parallel (\text{takes} :: t_4)^+ \\
&\quad\ \parallel (\text{tel} :: t_5 | \text{email} :: t_5), \\
\delta(t_3) &= (\text{teaches} :: t_4)^+, \\
\delta(t_4) &= \delta(t_5) = \varepsilon
\end{aligned}
$$

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

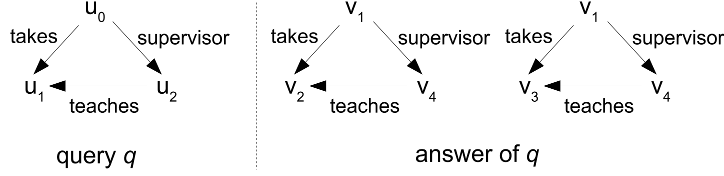

Figure 4: Query q and its answer over G.

(t_5 corresponds to string type). Here, consider S and the graph G in Fig. 3. In RBE, $a :: t$ matches an edge e if e is labeled by a and the target node of e is of type t. Thus we can verify that G is a valid graph of S.

ShEx has two semantics of typing: single-type typing and multi-type typing (Staworko et al., 2015). First, a *single-type typing* (*s-typing*) of G w.r.t. S is a function $\lambda : V \to \Gamma$ that associates every node $v \in V$ with a type $\lambda(v)$. Let

$$\textit{out-lab-type}_G^{\lambda}(v) = \{|a :: \lambda(v) \mid (v,a,v') \in E|\},$$

where $\{|\cdots|\}$ denotes a bag. Then λ is a *valid s-typing* if for every node $v \in V$, $\lambda(v)$ matches $\textit{out-lab-type}_G^{\lambda}(v)$. Second, a *multi-type typing* (*m-typing*) of G w.r.t. S is a function $\lambda : V \to 2^{\Gamma}$ that associates every node $v \in V$ with a set of types $\lambda(v)$. Let $\textit{Out-lab-type}_G^{\lambda}(v) = L(Flatten(\textit{out-lab-type}_G^{\lambda}(v)))$. For example, if $\textit{out-lab-type}_G^{\lambda}(v)) = \{|a :: \{t_1,t_2\}, b :: \{t_3\}|\}$, then $\textit{Out-lab-type}_G^{\lambda}(v)) = \{\{|a :: t_1, b :: t_3|\}, \{|a :: t_2, b :: t_3|\}\}$. Then an m-typing λ is *valid* if

1. $\lambda(v) \neq \emptyset$ for every $v \in V$, and
2. for every $v \in V$ and every $t \in \lambda(v)$, $\textit{Out-lab-type}_G^{\lambda}(v)) \cap \delta(t) \neq \emptyset$.

G is *valid* w.r.t. S under single-type (multi-type) semantics if there is a valid s-typing (m-typing) of G w.r.t. S.

3 DETECTING UNSATISFIABLE QUERY

To determine if a query q is unsatisfiable under a ShEx schema S, our algorithm checks if each node in q is safely matched by a type of S. To do this, we treat S as a graph, called *schema graph*. In the following, we first define schema graph and then present our algorithm.

3.1 Schema Graph

Let $S = (\Sigma, \Gamma, \delta)$ be a ShEx schema. Then the *schema graph* of S is denoted by a 4-tuple $G_S = (V_S, E_S, C^{or}, C^{int})$, where $V_S = \Gamma$ is a set of nodes (types), $E_S = \{(t,l,t') \mid \delta(t) \text{ includes } l :: t'\}$ is a set of edges, C^{or} is a collection of sets of "disjunctive" edges, and C^{int} is a collection of sets $C^{[n,m]}$ of edges associated with interval $[n,m]$. For example, consider the ShEx schema S presented in the previous section. Then Fig. 5 shows the schema graph $G_s = (V_s, E_s, C^{or}, C^{int} = \{C^*, C^+, C^?\})$ of S, where

$$\begin{aligned}
V_S &= \{t_1, t_2.t_3, t_4, t_5\}, \\
E_S &= \{(t_1, \text{student}, t_2), (t_2, \text{supervisor}, t_3), \\
&\quad (t_2, \text{takes}, t_4), (t_2, \text{tel}, t_5), (t_2, \text{email}, t_5), \\
&\quad (t_3, \text{teaches}, t_4)\}, \\
C^{or} &= \{\{(t_2, \text{tel}, t_5), (t_2, \text{email}, t_5)\}\}, \\
C^* &= \{(t_1, \text{student}, t_2)\}, \\
C^+ &= \{(t_2, \text{takes}, t_4), (t_3, \text{teaches}, t_4)\}, \\
C^? &= \{(t_2, \text{supervisor}, t_3)\}.
\end{aligned}$$

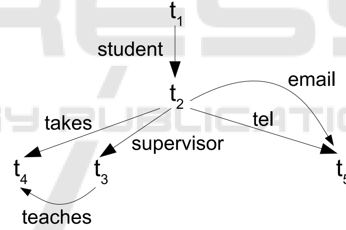

Figure 5: Schema graph of S.

Under s-type semantics, our algorithm uses the schema graph defined above. On the other hand, under m-typing semantics we need some modification to the schema graph, since some nodes may be associated with more than one type. To handle this, we create new types representing "combined types." Here, we briefly explain this by an example. For types t_1 and t_2, the *combined type* of t_1 and t_2, denoted t_{12}, is obtained as follows.

1. Find the common part of $\delta(t_1)$ and $\delta(t_2)$ without taking care of target type. For example, let $\delta(t_1) = a :: t_3 | b :: t_4$ and $\delta(t_2) = a :: t_2$. Then we obtain $a :: t_3$ from $\delta(t_1)$ and $a :: t_2$ from $\delta(t_2)$ since "a" is the common label. If the result is empty, t_{12} is not created.
2. For each corresponding "label::type" pair in $\delta(t_1)$ and $\delta(t_2)$, combine the target types of the "label::type" pair. In this example, since $a :: t_3$ of

$\delta(t_1)$ corresponds to $a :: t_2$ of $\delta(t_2)$, the target types t_2 and t_3 are combined and we obtain $\delta(t_{12}) = a :: t_{23}$.

3. Each edge incident to t_1 or t_2 is "copied" for t_{12}, e.g., if there is an edge (t, l, t_1), then we also have (t, l, t_{12}).

To summarize, under m-typing semantics we firstly obtain all the combined types as above, add the combined types and the "copied" edges incident to the combined types to S, and then apply our algorithm to the modified schema of S. Note that t_1 can be combined with t_2 only if $\delta(t_1)$ and $\delta(t_2)$ are "enough close" to each other in that a node v of type t_1 is also of t_2. Thus, in general the number of such combined types generated from a ShEx schema would be rather small.

3.2 Algorithm

Basically, our algorithm is based on solving the subgraph isomorphism problem between query q and schema graph of S. However, existing methods for solving the problem cannot be applied to our problem due to the following reasons. First, the schema graph may contain disjunction, and thus edges in the same set of C^{or} cannot be matched at the same time. Second, RBE may contain intervals, which imposes an upper bound on the number of outgoing edges and thus such a restriction needs to be taken into account when finding a match between edges of q and S. Third, more than one query node may have the same type; thus, the mapping from pattern nodes to schema types may not be bijective.

Our algorithm consists of two parts: Algorithms 3.1 and 3.2. Algorithm 3.1 is the initialization part of our algorithm and Algorithm 3.2 is called from Algorithm 3.1 to find a match between q and S. First, Algorithm 3.1 works as follows. Line 1 initializes M, which is a set to store "answer", i.e., a set of pairs of matched nodes between q and schema graph G_S. MAKESCHEMAGRAPH on line 2 creates the schema graph G_S of S. Lines 3 to 6 find, for each node u of q, a set of candidate nodes (types) $C(u) \subseteq \Gamma$. Here, $C(u)$ consists of "candidate" types $t \in \Gamma$ such that the set of outgoing labels of t includes that of u. FILTERCANDIDATES on line 4 is a function that computes $C(u)$. On line 7, FINDMATCH recursively traverses q and G_S and finds "matching" between q and G_S.

Algorithm 3.2, called FINDMATCH, recursively traverses q and G_S and adds matched pairs to M. Lines 1 and 2 check whether the size of M reaches the number of nodes of q. If it holds, then every node in q is safely matched by a type of S and thus output

Algorithm 3.1: Unsatisfiability CHECKING.

Input: query $q = (V(q), E(q))$, ShEx schema $S = (\Sigma, \Gamma, \delta)$
Output: "satisfiable" or "unsatisfiable"

```
1: M := ∅;
2: G_S = MAKESCHEMAGRAPH(S);
3: for each u ∈ V(q) do
4:     C(u) := FILTERCANDIDATES(q, G_S, u);
5:     if C(u) = ∅ then
6:         return "unsatisfiable";
7: FINDMATCH(q, G_S, M, nil)
```

Algorithm 3.2: FINDMATCH.

Input: query $q = (V(q), E(q))$, schema graph $G_S = (\Gamma, E_S, C^{or}, C^{int})$, set $M \subseteq V(q) \times \Gamma$, current node u_c
Output: "satisfiable" or "unsatisfiable"

```
1: if |M| = |V(q)| then
2:     return "satisfiable";
3: else
4:     u := NEXTVERTEX(q, u_c);
5:     E_u ← {(u, l, u') ∈ E(q) | u' appears in M}
              ∪ {(u', l, u) ∈ E(q) | u' appears in M}
6:     E_t ← {(t, l, t') ∈ E_S | t' appears in M}
              ∪ {(t', l, t) ∈ E_S | t' appears in M}
7:     for each t ∈ C(u) such that t is not in M do
8:         if CHECKDISJUNCTION(u, t, q, G_S, E_u, E_t)
           then
9:             if ISUSABLEONCE(u, t, q, G_S, E_u, E_t)
               then
10:                if ISUSABLENTIMES(u, t, q, G_S, E_u, E_t)
                   then
11:                    UPDATESTATE(M, u, t);
12:                    FINDMATCH(q, G_S, M, u);
13:                    RESTORESTATE(M, u, t);
14:    return "unsatisfiable";
```

"satisfiable". NEXTVERTEX on line 4 is a function that returns the "next" node $u \in V(q)$ of the current node u_c. If $u_c = nil$, then the function returns the first node of q. Here, we assume that there is some order on $V(q)$ (the order can be arbitrary), and NEXTVERTEX works based on that order. Lines 7 to 10 determine, for each candidate $t \in C(u)$, whether t matches u w.r.t. M. Let E_u be the set of edges between u and the q's nodes in M, and let E_t be the set of edges between t and the G_S's nodes in M. Checking if t safely matches u is done by the following three functions.

1. CHECKDISJUNCTION on line 8 checks if t can match u without violating the disjunction constraints in S. This is done by comparing E_u and E_t along with C^{or} of S. For example, if t has ex-

actly two outgoing edges (t,a,t') and (t,b,t'') that are in the same set of C^{or} while u has two outgoing edges (u,a,u') and (u,b,u'') such that t' (t'') matches u' (resp., u''), then t cannot match u.

2. ISUSABLEONCE on line 9 checks if t can match u without violating the constraints on the edge cardinality of q except C^{int}. That is, if an edge incident to t does not appear in C^{int}, then the corresponding edge incident to u must be exist at most once. The function checks if the condition holds for u and t.
3. ISUSABLENTIMES on line 10 checks if t can match u without violating C^{int}. For example, suppose that $(t,a,t') \in C^{[0,k]}$. Then the number of edges in E_u that match (t,a,t') must be no more than k. The function checks such a condition by comparing E_u and E_t along with C^{int}.

If all the above checks are passed, then pair (u,t) is added to M by UPDATESTATE on line 11. On line 12, call FINDMATCH recursively in order to find matches for the rest of nodes of q and G_S. If no answer is found by the call of FINDMATCH, then RESTORESTATE on line 13 restores M, i.e., (u,t) is deleted from M, and back to line 7. Finally, if no answer is found until all the nodes in q are examined, output “unsatisfiable” on line 14.

Finally, consider briefly the computational complexity of the problem. In theory, the problem cannot be solved efficiently.

Theorem 1. *Detecting unsatisfiable pattern queries is NP-hard under both s-typing and m-type semantics.*

The algorithm checks if each node u of q is matched by a node t of S. Thus the time complexity may become exponential in the worst case, but which is unavoidable due to the above theorem. However, the size of schema is much smaller than that of data graph, and the algorithm terminates as soon as one satisfiable matching is found. Thus, although the problem is NP-hard, the algorithm can be executed highly efficiently as shown in the next section.

4 PRELIMINARY EXPERIMENTS

We conducted preliminary experiments to evaluate our algorithm. To detect unsatisfiable queries, the algorithm has to be executed before executing queries over RDF data. Therefore, we need to verify that the execution time of our algorithm is enough small compared with query execution time over RDF data. The algorithm was implemented in Ruby 2.5.1, and all the experiments were executed on a machine with Intel(R) Core(TM) m3-7Y30 CPU 1.60GHz, 4.00GB RAM, Windows 10 Home OS.

We used two datasets. The first one was generated by SP^{2}Bench (Schmidt et al., 2008) and the second one was generated by BSBM (Bizer and Schultz, 2009). SP^{2}Bench is a well-known SPARQL performance benchmark tool based on DBLP. For the SP^{2}Bench dataset, we generated RDF data of size 1,087,517 byte (10,291 triples) and 5,400,376 byte (50,168 triples). Since SP^{2}Bench does not have any ShEx schema, we manually created a ShEx schema (type: 11, edge: 69) based on (Schmidt et al., 2008). BSBM is also a well-known SPARQL performance benchmark tool, which data is based on e-commerce use case. For the BSBM dataset, we generated RDF data of size 2,583,293 byte (10,250 triples) and 10,216,303 byte (40,377 triples). BSBM does not have any ShEx schema either, therefore we created a ShEx schema (node: 10, edge: 71) based on (Bizer and Schultz, 2009). Since both of SP^{2}Bench and BSBM assume s-type semantics implicitly, the experiments were conducted under s-typing semantics.

As for pattern queries, we made a Ruby program for generating queries. In short, this program randomly selects labels and types from a given ShEx schema and generates nodes and edges, and then the authors check unsatisfiability of the generated queries manually. We generated 50 different unsatisfiable queries (10 queries for each of 5 different query sizes) for each dataset. We also made a Ruby program to execute queries based on the Ullmann’s algorithm (Ullmann, 1976). Note that, although a number of algorithms for pattern matching are proposed, the data used in this experiments are very small and thus which algorithm is used hardly affects execution time. Actually, in such a case “preprocessing” such as reading data and registering nodes and edges into lists/arrays accounts for most portion of execution time, which is common to any kind of pattern matching algorithms.

Tables 1 and 2 show the results. All the execution times were measured in seconds. Each query execution time in the tables is the average of those of 10 queries. As shown in the tables, compared with the query execution time over RDF data, the execution time of our algorithm is much smaller and, e.g., we can save about 300 seconds for the larger data of SP^{2}Bench dataset. Also, the ratio values are almost negligible. Note that, since the size of RDF data used in the experiment is rather small, the ratio would become much smaller if we use larger RDF data. Therefore, if a user tries to execute a query and it is unsatisfiable, our algorithm can save a lot of time by detecting the unsatisfiability of the query. And even if it is

Table 1: Result for SP2Bench dataset.

query size	3	4	5	6	7
(a) Unsatisfiability Checking time (our algorithm)	0.00343	0.00357	0.00501	0.00463	0.00469
(b) Query execution time (10,291 triples)	43.2	34.1	34.1	36.0	38.9
(c) Query execution time (50,168 triples)	369	343	396	406	339
Time ratio (a/b)	0.0000793	0.000105	0.000147	0.000129	0.000120
Time ratio (a/c)	0.00000927	0.0000104	0.0000126	0.0000114	0.0000138

Table 2: Result for BSBM dataset.

query size	3	4	5	6	7
(a) Unsatisfiability checking time (our algorithm)	0.0105	0.00737	0.00888	0.0114	0.00762
(b) Query execution time (10,250 triples)	18.1	15.5	19.9	23.5	17.6
(c) Query execution time (40,377 triples)	224	216	236	230	238
Time ratio (a/b)	0.000574	0.000475	0. 000446	0.000486	0. 000433
Time ratio (a/c)	0.0000469	0.0000342	0. 0000377	0. 0000497	0.0000321

satisfiable, unsatisfiability checking can be done very quickly and little time is wasted.

5 CONCLUSIONS

In this paper, we proposed an algorithm for detecting unsatisfiable pattern queries under ShEx schemas. Experimental results suggest that our algorithm run efficiently w.r.t. the running time of query execution.

As future issues, since the experiments were conducted under s-type semantics only, we need to conduct experiments under m-typing semantics. We also need to use other datasets under variety kinds of of ShEx schemas. Moreover, ShEx has more functions not discussed in this paper (e.g., negation). Thus we need to consider extending our algorithm to adopt such functions.

REFERENCES

Baker, T. and Prud'hommeaux, E. (2019). Shape expressions (ShEx) primer. http://shexspec.github.io/primer/.

Benedikt, M., Fan, W., and Geerts, F. (2008). XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2):8:1–8:79.

Bizer, C. and Schultz, A. (2009). The berlin SPARQL benchmark. In *International journal on Semantic Web and information systems 5(2)*, pages 1–24.

Cristani, M., Bertolaso, A., Scannapieco, S., and Tomazzoli, C. (2018). Future paradigms of automated processing of business documents. *International Journal of Information Management*, 40:67–75.

Figueira, D. (2018). Satisfiability of XPath on data trees. *ACM SIGLOG News*, 5(2):4–16.

Geneves, P., Layada, N., and Knyttl, V. (2011). XML reasoning solver user manual. available from https://hal.inria.fr/inria-00339184v2/document.

Groppe, J. and Groppe, S. (2007). Filtering unsatisfiable XPath queries. *Data & Knowledge Engineering*, 64(1):134–169.

Ishihara, Y., Suzuki, N., Hashimoto, K., Shimizu, S., and Fujiwara, T. (2013). 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)*.

Montazerian, M., Wood, P. T., and Mousavi, S. R. (2007). XPath query satisfiability is in PTIME for real-world DTDs. In *Proc. International XML Database Symposium*, pages 17–30.

Schmidt, M., Hornung, T., Lausen, G., and Pinkel, C. (2008). SP2Bench: a SPARQL performance benchmark. In *Proc. ICDE*, pages 371–393.

Staworko, S., Boneva, I., Labra Gayo, J. nad Hym, S., Prud'hommeaux, E., and Sorbrig., H. (2015). Complexity and expressiveness of ShEx for RDF. In *Proceedings of 18th In-ternational Conference on Database Theory (ICDT 2015)*, pages 195–211.

Thornton, K., Solbrig, H., Stupp, G. S., Labra Gayo, J. E., Mietchen, D., Prud'hommeaux, E., and Waagmeester, A. (2019). Using shape expressions (ShEx) to share

RDF data models and to guide curation with rigorous validation. In Hitzler, P., Fernandez, M., Janowicz, K., Zaveri, A., Gray, A. J., Lopez, V., Haller, A., and Hammar, K., editors, *In Proceedings of the European Semantic Web Conference*, pages 606–620.

Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42.

Zhang, X., den Bussche, J. V., and Picalausa, F. (2016). On the satisfiability problem for SPARQL patterns. *Journal of Artificial Intelligence Research*, 55:403–428.