Tiako P.F. (ed) *Automated Systems, Data, and Sustainable Computing*. Chronicle of Computing. OkIP.
© 2022 Oklahoma International Publishing https://doi.org/10.55432/978-1-6692-0001-7_2

Detecting Unsatisfiable Conjunctive Property Path under Shape Expression Schema

Yuuki Maeda[a] and Nobutaka Suzuki[a]*

[a]*University of Tsukuba, 1-2 Kasuga, Tsukuba, 305-8550, Japan*

ABSTRACT

Shape Expression (ShEx) is a novel schema language proposed for RDF/graph data. For a ShEx schema S and a query q, q is said to be unsatisfiable if for any valid data D under S q reports an empty answer over D. In general, the size of RDF/graph data is very large, and thus it is inefficient to perform unsatisfiable queries on such data. Therefore, it is desirable that we can detect unsatisfiable queries efficiently before executing them. In this paper, we consider Conjunctive Property Path, a generalization of Property Path defined in SPARQL 1.1, as the query language. First, we propose an algorithm for determining satisfiability of Conjunctive Property Path queries under ShEx schema. Then we conducted a preliminary experiment, which results suggest that the proposed algorithm determines if a given Conjunctive Property Path query is satisfiable efficiently.

Keywords: graph data, RDF, Property Path, satisfiability

I. INTRODUCTION

For over years, RDF/graph is a popular data format widely used for variety of areas, Linked Open Data, social graph, biological network, and so on. In general, the sizes of such data tend to be very large, and thus it is useless and time-consuming to perform unsatisfiable queries over the data. Here, for a query q and a schema S, q is *unsatisfiable* under S if for any valid data D under S q reports an empty answer over D. Clearly, it is desirable that we can detect unsatisfiable queries efficiently before executing them.

In this paper, we consider a query satisfiability problem under Shape Expression (ShEx). Here, ShEx is a novel schema language for RDF data proposed by W3C Draft Community Group (Baker, T. and Prud'hommeaux, 2019), and it is already used in various areas (Thornton et al, 2019). ShEx shares a number of core features with another novel schema language Shapes Constraint Language (SHACL), therefore the result in this paper can be applied to SHACL as well. As for query language, we use Conjunctive Property Path (CPP). Here, Property Path is a well-known path query language whose specification is given in SPARQL 1.1, and a CPP query consists of Property Path queries connected by conjunctive operators.

In this paper, we fist show that determining whether a CPP query is satisfiable under a given ShEx schema is NP-hard. Then we consider a slight restriction of CPP, acyclic CPP, and propose a polynomial-time algorithm for determining whether a given acyclic CPP query is satisfiable under a given ShEx schema. In short, the algorithm decomposes a given CPP query q into Property Path queries q_1, q_2, … , q_n, and then checks satisfiability of whole expression q by iteratively refineing the types of S for which q_i is satisfiable. We conducted a preliminary experiment, which results suggest that the proposed algorithm can determine whether a given CPP query is satisfiable efficiently.

* Corresponding author E-mail: nsuzuki@slis.tsukuba.ac.jp

RELATED WORK

Query satisfiability has been a popular problem in database management field. For example, a number of studies on XPath satisfiability under DTD or XML Schema have been made, e.g., (Benedikt & Fan & Geerts, 2008; Figueira, 2018). However, XML is modeled as ordered tree while RDF data is modeled as unordered graph, and thus the studies cannot apply to RDF/graph data. Zhang et al. consider satisfiability of SPARQL pattern query without schema (Zhang et al., 2016). Matsuoka et al. propose an algorithm for checking satisfiability of pattern queries under ShEx schema (Matsuoka & Suzuki, 2020), where the pattern query is a small fragment of our CPP query. To the best of the authors' knowledge, no studies on satisfiability of CPP queries under ShEx schemas have been made so far.

II. PRELIMINARIES

Let Σ be a set of labels. A *graph* G 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 labeled directed edges.

Unlike XML data model, in RDF/graph data model the order among sibling nodes is less significant. Thus ShEx uses *regular bag expression* (RBE) to represent node type (Staworko et al., 2015) instead of regular expression. RBE is defined similar to regular expressions except that RBE uses unordered concatenation instead of ordered one. 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 unordered concatenation.
- 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 is an RBE, then $r^?$, r^*, r^+ are RBEs.

RBE r is *disjunction-capsuled* if every disjunctive expression $r_1 | r_2 | \cdots | r_k$ in r is enclosed by + or *. 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 $\Sigma \times \Gamma$. S is *disjunction-capsuled* if $\delta(t)$

is disjunction-capsuled for every $t \in \Gamma$. For example, let $S = (\Sigma, \Gamma, \delta)$ be a ShEx schema, where Σ= {takes, supervisor, teaches}, $\Gamma = \{t_1, t_2, t_3\}$, $\delta(t_1)$=(takes::t_2)*||(supervisor::t_3)?, $\delta(t_2)= \varepsilon$, $\delta(t_3)$=(teaches::t_2)*. Then it is easy to verify that the graph in Figure 1 is valid for S since each node conforms to the type in color red.

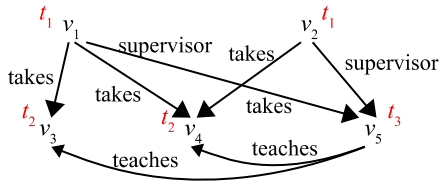

Figure 1: example of valid graph

A *Property Path* over Σ is recursively defined as follows:

- $\varepsilon, a \in \Sigma$, and * are Property Paths.
- For a set $\{a_1, a_2, \cdots, a_k\}$ of labels, $!\{a_1, a_2, \cdots, a_k\}$ is a Property Path that matches any label except $a_1, a_2, \cdots, a_k$.
- For any $a \in \Sigma$, a^{-1} is a Property Path, where -1 denotes inverse.
- If $r_1, r_2, \cdots, r_k$ are Property Paths, then $r_1.r_2.\cdots.r_k$ and $r_1|r_2|\cdots|r_k$ are Property Paths.
- If r is a Property Path, then $r^?$, r^*, r^+ are Property Paths.

A *Conjunctive Property Path query* (*CPP query*, for short) is constructed by Property Paths with head and tail variables connected by conjunctive operators. Formally, CPP query Q is defined as follows:

$$Q = \wedge_i (x_i, p_i, y_i),$$

where x_i, y_i are variables and p_i is a Property Path. (x_i, p_i, y_i) is *evaluated* true over graph G if for some assignment $x_i \leftarrow v$ and $y_i \leftarrow v'$, G contains a path from v to v' such that p_i matches the sequence of labels on the path. By Var(Q) we mean the set of variables occurring in Q. Let $G(Q)$ be the graph obtained by regarding each variable and each CPP query (x_i, p_i, y_i) as a node and an edge from x_i to y_i labeled by "p_i", respectively. For query (edge) (x_i, p_i, y_i), x_i is called *head* and y_i is called *tail*. We say that a CPP query Q is *acyclic* if $G(Q)$ is acyclic.

For a ShEx schema S and a CPP query Q, Q is *satisfiable* under S if for some valid graph G of S, there is an assignment of Var(Q) under which Q is evaluated true over G.

III. ALGORITHM

In this section, we first show the complexity of the satisfiability problem briefly, then give an algorithm for solving the problem. We have the following theorem.

Theorem 1: For a ShEx schems S and a CPP query Q, determining whether Q is satisfiable under S is NP-hard.

Thus, in the following we impose some restrictions on S and Q; we assume that S is disjunction-capsuled and that Q is acyclic. Under the restrictions, we present a polynomial-time algorithm for determining whether Q is satisfiable under S.

In the algorithm, ShEx schema $S = (\Sigma, \Gamma, \delta)$ is treated as an NFA. Formally, the NFA of S is denoted $M_S = (\Gamma, \Sigma, \Delta, nil, \Gamma)$, where Γ is a set of states (types), Δ is a transition function such that $\Delta(t,a) = \{t' \mid a::t'$ appears in $\delta(t)\}$ for any $t \in \Gamma$ and any $a \in \Sigma$, the start state is *nil* (the start state is specified during execution of the algorithm), and the set of final states is Γ.

Let $Q(x)$ be the set of Property Path queries in Q whose head is x, that is,

$$Q(x) = \{(x_i, p_i, y_i) \mid x_i = x \text{ and } (x_i, p_i, y_i) \text{ appears in } Q\}.$$

Let T:Var(Q) $\rightarrow 2^\Gamma$ be an assignment to the variables in Q. During execution of the algorithm, $T(x)$ holds "valid" types for x at each iteration; $T(x)$ initially holds all the types of ShEx schema S, and is gradually reduced until $T(x)$ converges. The *one-step refinement* of $Q(x)$ under T over M_S is a tuple $(X, Y_1, \ldots, Y_n)$ of sets of types satisfying the following conditions:

1. $X \subseteq T(x)$ and $Y_i \subseteq T(y_i)$ for every $1 \le i \le n$.
2. For every $t \in X$, for every $1 \le i \le n$, and for every $t' \in Y_i$, t' is reachable from t via p_i over M_S.

We now show the algorithm. We assume that there is a partial order on Var(Q); x $\succ$ y if $(x, p, y) \in Q$.

Input: ShEx schema $S = (\Sigma, \Gamma, \delta)$, CPP query Q
Output: satisfiable or unsatisfiable
1. Let M_S be the NFA of S
2. $T(x) \leftarrow \Gamma$ for every x $\in \Gamma$
3. Let $G(Q)$ be the graph obtained from Q
4. Let $x_1,\ldots,x_n$ be the nodes (variables) in $G(Q)$ having no incoming edges
5. Let PQ be a priority que over Var(Q)
6. Add $x_1,\ldots,x_n$ to PQ
7. **while** PQ is nonempty **do**
8. $x \leftarrow$ dequeue(PQ)
9. Let $(x, p_1, y_1), \ldots, (x, p_n, y_n)$ the Property Path queries in $Q(x)$. Compute the one-step refinement of $Q(x)$ under T over M_S. Let $(X, Y_1, \ldots, Y_n)$ be the result.
10. **if** one of $X, Y_1, \ldots, Y_n$ is empty **then**
11. **return** unsatisfiable
12. $T(x) \leftarrow X$, $T(y_1) \leftarrow Y_1, \ldots, T(y_n) \leftarrow Y_n$
13. **for** each $z \in \{x, y_1, \ldots, y_n\}$ **do**
14. **if** $(z = x$ and $X \subsetneq T(x))$ or $(z = y_i$ and $Y_i \subsetneq T(y_i))$ **then**
15. **if** $z != x$ **then**
16. Add z to PQ
17. Let $(w_1, p_1, z), \ldots, (w_k, p_k, z)$ be the incoming edges of z in $G(Q)$ s.t. $w_i != x$
18. Add $w_1, \ldots, w_k$ to PQ
19. **return** satisfiable

We use priority queue PQ over Var(Q), where the priority is determined by the partial order $\succ$ over Var(Q) (ties are broken arbitrarily). PQ holds variables that should be evaluated. Initially, PQ holds $x_1,\ldots,x_n$ in $G(Q)$ having no incoming edges (lines 4 to 6). Then we enter the while loop to evaluate each variable (lines 7 to 18). A variable x is picked up from PQ and then edges whose head is x are evaluated (line 9). The one-step refinement is done by traversing states of M_S (details are omitted due to space limitation). If some variable becomes empty, then Q is unsatisfiable and the result is reported (lines 10 and 11).

Otherwise, the results of the one-step refinement are assigned to T (line 12). If there is a variable z whose types are reduced, i.e., $X \subsetneq T(x)$ or $Y_i \subsetneq T(y_i)$, then we have to re-evaluate z and the variables incident to z. Thus, z and head w_i of edges having z as the tail are added to PQ (lines 17 and 18). Finally, if the evaluation converges without any variables being empty, then it is reported that Q is satisfiable (line 19).

The algorithm runs in polynomial time since (1) the one-step evaluation can be done in polynomial time by traversing M_S and (2) each $T(x)$ has at most $|\Gamma|$ types and for each iteration of the while loop at least one type is deleted from some $T(x)$.

IV. Experimental Results

We conducted a preliminary experiment on the algorithm. All experiments were executed on a machine with Intel Core i5 CPU 2.3 GHz dual-core, 8.00GB RAM, MacOS Big Sur 11.4.

In the experiment, we need unsatisfiable CPP queries. Thus, we made a Ruby program that automatically generates CPP queries by randomly combining multiple labels and then randomly concatenates them. Length (i.e., the number of Property Paths) of CPP query, length of Property Paths, shape of CPP query (straight or DAG), and probability of assigning operator (?, *, +) to label can be specified.

We implemented the algorithm using Ruby, and generated N3 data by SP^{2}Bench (Schmidt et al., 2008) consisting of 10000 triples (1.64MB). We also created a ShEx schema for SP^{2}Bench. Using the ShEx schema, the CPP queries, and the data, we first executed the proposed algorithm to check satisfiability of the CPP queries under the ShEx schema. We also executed the CPP queries over the data. Each CPP query was processed by Ruby SPARQL library.

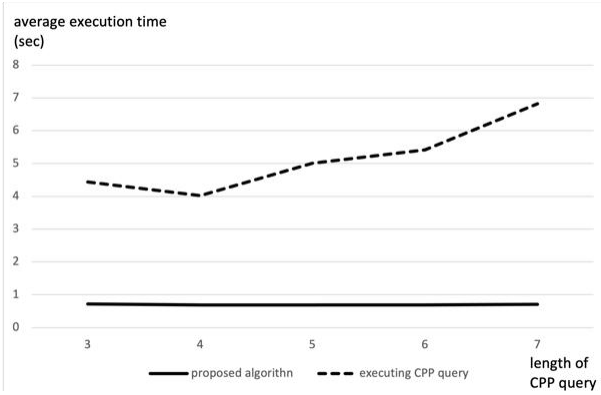

Figure 2: experimental result of simpler case

Figures 2 and 3 show the results. Figure 2 shows a simpler case, where each CPP query has no operator, each CPP query is of length 3 to 7, the shape of each CPP query is straight, and the length of each Property Path is 10 (i.e., each CPP query is of length 30 to 70). Here, for a CPP query $Q = \wedge_i (x_i, p_i, y_i)$, we say that Q is *straight* if $y_i = x_{i+1}$ for every i. Figure 3 shows a more general case, where an operator is assigned to a label with probability 5%, each CPP query consists of 3 to 7 Property Paths, the shape of each CPP query is DAG, and the length of each Property Path is 5. In both cases, for each length of CPP query, we generated 10 different CPP queries and measured the average execution time of them.

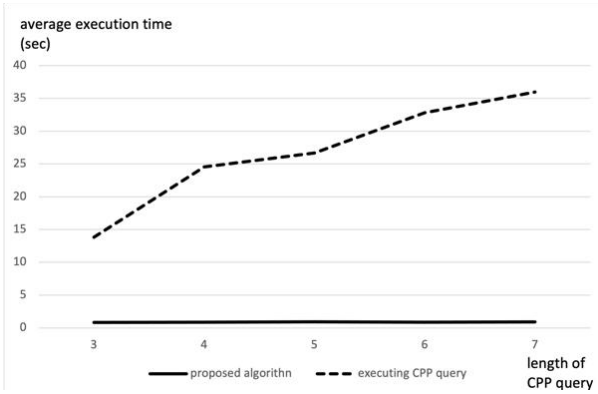

Figure 3: experimental result of more general case

From these results, we can see that the proposed algorithm can detect unsatisfiable queries more efficiently, compared to performing CPP queries over the RDF data. This means that by detecting unsatisfiable CPP queries before execution, we can safely save time for executing CPP queries, and even if executed query is satisfiable, the time loss for running the algorithm is very limited. In addition, in the latter case, the execution time to perform CPP queries increases significantly, while the increase in execution time of the proposed algorithm is negligible.

V. Conclusion

In this paper, we proposed an algorithm for determining satisfiability of CPP queries under ShEx. The results of experiments suggest that we can save time by detecting unsatisfiable CPP queries before executing them. As a future work, we need to consider cyclic CPP queries and improving the efficiency of the algorithm. We plan to extend the algorithm to solve these issues.

Acknowledgment

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

References

Baker, T. and Prud'hommeaux, E. (2019). Shape Expression (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.

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

Matsuoka, S. and Suzuki, N. (2020). Detecting unsatisfiable pattern queries under shape expression schema, Proceedings of the 16th International Conference on Web Information Systems and Technologies, 285 – 291.

Schmidt, M., Hornung, T., Lausen, G., and Pinkel, C. (2008). SP2Bench: a SPARQL performance bench- mark. Proceedings of ICDE, 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. Proceedings of 18th International Conference on Database Theory, 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, Proceedings of the European Semantic Web Conference, pages 606–620.

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.