IEICE TRANS. INF. & SYST., VOL.E99–D, NO.4 APRIL 2016

PAPER Special Section on Data Engineering and Information Management

An Algorithm for All-Pairs Regular Path Problem on External Memory Graphs*

Nobutaka SUZUKI[†a)], ***Member***, **Kosetsu IKEDA**[††], ***Student Member***, ***and*** **Yeondae KWON**[†††], ***Nonmember***

SUMMARY In this paper, we consider solving the all-pairs regular path problem on large graphs efficiently. Let G be a graph and r be a regular path query, and consider finding the answers of r on G. If G is so small that it fits in main memory, it suffices to load entire G into main memory and traverse G to find paths matching r. However, if G is too large and cannot fit in main memory, we need another approach. In this paper, we propose a novel approach based on external memory algorithm. Our algorithm finds the answers matching r by scanning the node list of G sequentially. We made a small experiment, which suggests that our algorithm can solve the problem efficiently.

key words: *graph data, regular path query*

1. Introduction

Graph is being used in broad areas such as social network, life science, Semantic Web, and so on. Recently, such graphs are greatly increasing and their sizes are rapidly growing. Regular path query is a popular query language for such graphs and is being studied actively so far. In this paper, we consider processing regular path queries efficiently on large graphs. In particular, we focus on solving the all-pairs regular path problem, which is to find all pairs (n, n') of nodes such that there is a path between n and n' whose sequence of labels matches a given regular path query.

Let us consider finding the answers of a regular path query r on a graph G. If G is so small that it can be processed in main memory, it suffices to load entire G into main memory, transform r into an NFA A, and find the "answer" paths of G whose sequence of labels matches A. However, if G is too large to be processed in main memory, the above approach is not applicable to obtaining the answers of r. Thus we need another approach to solving the problem on large graphs efficiently.

A possible approach to this problem is to store a graph in some graph store and issue regular path queries on the graph store. However, to solve the problem we have to determine, for a large number of pair of nodes (n, n'), whether there exists a path between n and n' matching a given regular path query. This requires fetching nodes and edges on a disk a large number of times, and thus the approach is not necessarily efficient for solving the problem.

Another possible approach is to use the SPARQL query language [2]. In fact, SPARQL is added support for regular path query processing called "Property Path" since version 1.1, and thus we can use a regular path query on RDF stores supporting SPARQL 1.1. However, Property Path is under "simple walk semantics" (no node appears twice on a path except the start and the end of the path), and thus Property Path evaluation is quite inefficient even for very restricted queries [3], e.g., for nodes n, n' and a query $q = (aa)^*$, determining whether there exists a path from n to n' that matches q is NP-complete.

To cope with these problems, we propose another approach based on external memory algorithm. In order to find answer paths on a graph efficiently, our approach is based on scanning the graph sequentially rather than fetching nodes/edges each time they are required. Let G be a graph (file) and S be a fixed area in main memory to load a subgraph of G. Firstly, our algorithm scans G sequentially and repeats the following until EOF is found.

1. Load data of size $|S|$ from G into S (thus the loaded data is a subgraph of G) and find answers that can be obtained by traversing S.
2. During the traversal of S, find connections among paths inside and outside S, and store them into another graph called "contracted graph".

After the sequential scan, the algorithm traverses the contracted graph and finds the rest of answers whose path runs across different subgraphs, which is also done by a sequential scan.

Related Work

There are a number of studies related to regular path queries, e.g., [4], [5] are extensive surveys on regular path queries and related query languages. However, studies on regular path query processing on large graphs are unexpectedly not many. [6] proposes an algorithm for solving all-pairs regular path problem efficiently on large graphs. This algorithm is an in-memory algorithm and assumes that a graph fits in main memory. [7]–[10] propose distributed approaches for regular path query processing. These are suitable for graphs inherently distributed over multiple machines or graphs that

Manuscript received June 29, 2015.
Manuscript revised November 9, 2015.
Manuscript publicized January 14, 2016.

[†]The author is with Faculty of Library, Information and Media Science, University of Tsukuba, Tsukuba-shi, 305–8550 Japan.
[††]The author is with Graduate School of Library, Information and Media Studies, University of Tsukuba, Tsukuba-shi, 305–8550 Japan.
[†††]The author is with Graduate School of Agricultural and Life Sciences, The University of Tokyo, Tokyo, 113–8657 Japan.

*This paper is a revised version of [1].

a) E-mail: nsuzuki@slis.tsukuba.ac.jp

DOI: 10.1587/transinf.2015DAP0018

are too large to be handled in a single machine. On the other hand, for graphs that can be handled in a single machine, we believe that our approach is a reasonable choice for regular path query processing.

A number of graph stores have been proposed, e.g., Pregel [11], Sparksee[†] (formerly known as Dex [12]), Grail [13], and Neo4j[††], and for some of them a powerful query language that can simulate regular path queries is available (Gremlin[†††]). However, these are not necessarily suitable for the all-pairs regular path problem on large graphs, since solving the problem on a graph store requires fetching a large number of nodes and edges due to the high complexity of the problem. gStore [14] is an RDF store with an efficient query processor, but this does not support regular path query. GraphChi [15] and TurboGraph [16] are systems designed for large-scale graph computation in a single PC. GraphChi uses a novel method called Parallel Sliding Window (PSW) for reducing non-sequential accesses to a disk. However, as remarked in [15], the PSW method is not suitable for efficient graph traversal. TurboGraph is designed to exploit full I/O and CPU parallelism and full overlap of CPU and I/O processing. However, the system does not support labeled graphs, and it is not obvious whether TurboGraph can be extended to deal with labeled graphs and solve the all-pairs regular path problem efficiently.

Many studies have been made for optimizing path queries on semistructured data and XML. For example, [17] proposes a numbering scheme for efficient regular path computation, and [18] and [19] propose additional index structures. However, these studies are designed for tree structured data and cannot be applied to non-tree graphs. [20] proposes optimization techniques for regular path queries by using graph schemas. However, most of current graphs have no explicit schema, and thus the techniques cannot be applied to such graphs.

SPARQL did not support regular path query until SPARQL 1.1, and thus a number of query language and extensions to SPARQL for supporting regular path query have been proposed. [21] proposes a query language called PSPARQL supporting regular path query. However, the paper provides only an implementation based on an in-memory algorithm. [22] proposes the GREEN system, which extends SPARQL so that it can handle regular path queries. However, the system is implemented as an extension to the ARQ library, which means that its query processing is done by iterative fetching of nodes and edges. [23] also proposes an extension of SPARQL for regular path query, but presents no specific query processing method. [24] proposes an RDF query language supporting regular path query. However, it is reported that the implementation of the language works on only small graphs [6]. [25] proposes another RDF query language SPARQLeR that supports regular path query. However, [25] reports that several tens of seconds is needed even for evaluating a single-source single-destination query on 6.6 million RDF statements, which implies that it would take significantly more time to solve the all-pairs regular path problem on larger graphs. In 2013, SPARQL 1.1 is published as W3C recommendation. SPARQL 1.1 supports regular path query by virtue of Property Path, and thus we can use regular path queries on RDF stores supporting SPARQL 1.1, e.g., Apache Jena[††††] and Sesame[†††††]. However, Property Path of SPARQL 1.1 is under simple walk semantics, and thus the complexity of regular path query evaluation is NP-hard even for very restricted queries [3]. Therefore, such RDF stores are not necessarily suitable for solving the all-pairs regular path problem on large graphs either.

Several external memory algorithms have been proposed in database research field, e.g., graph triangulation [26], strongly connected components [27], graph reachability [28], and k-bismulation [29]. To the best of our knowledge, however, no external memory algorithm for processing the all-pairs regular path problem has been proposed so far.

This paper is organized as follows. Section 2 gives preliminary definitions. Section 3 presents an external memory algorithm for processing regular path queries. Section 4 shows the correctness of the algorithm. Section 5 presents the I/O and CPU costs of the algorithm. Section 6 gives experimental results. Section 7 summarizes the paper.

2. Definitions

Let Σ be a set of labels. A *labeled directed graph* (*graph* for short) over Σ is denoted $G = (V, E, \Sigma)$, where V is a set of nodes and E is a set of labeled edges. Let $n, n' \in V$ be nodes. An edge from n to n' labeled by $l \in \Sigma$ is denoted $n \xrightarrow{l} n'$. A path p from n to n' is denoted $n \rightsquigarrow_p n'$. By $l(p)$ we mean the sequence of labels on p. For example, if $p = n_1 \xrightarrow{a} n_2 \xrightarrow{b} n_3 \xrightarrow{c} n_4$, then $l(p) = abc$.

A *regular path query* is defined as a regular expression over Σ, as follows.

- ε, $\emptyset$, and $a \in \Sigma$ are regular path queries.
- If $r_1, r_2, \cdots, r_n$ are regular path queries, then $r_1 r_2 \cdots r_n$ and $r_1 | r_2 | \cdots | r_n$ are regular path queries.
- If r is a regular path query, then r^* is a regular path query.

r^+ and $r?$ are abbreviations for $r^* r$ and $r|\varepsilon$, respectively. By $L(r)$ we mean the *language* of a regular path query r.

A nondeterministic finite automaton (NFA) over Σ is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of *states*, $q_0 \in Q$ is the *start state*, $F \subseteq Q$ is a set of *accepting states*, and $\delta : Q \times \Sigma \to 2^Q$ is a *transition function*. The *extension* of δ, denoted $\hat{\delta}$, is defined as follows.

- For any $q \in Q$, $\hat{\delta}(q, \varepsilon) = \{q\}$.

[†]http://www.sparsity-technologies.com/
[††]http://www.neo4j.org/
[†††]https://github.com/tinkerpop/gremlin
[††††]https://jena.apache.org/
[†††††]http://openrdf.org/

- For any $w \in \Sigma^*$ and any $a \in \Sigma$, $\hat{\delta}(q, wa) = \bigcup_{q' \in \hat{\delta}(q,w)} \delta(q', a)$.

That is, $q' \in \hat{\delta}(q, w)$ iff A enters state q' when A reads w in state q. The *language* accepted by A is defined as $L(A) = \{w \in \Sigma^* \mid q_f \in \hat{\delta}(q_0, w), q_f \in F\}$.

In this paper, we consider finding, for a graph $G = (V, E, \Sigma)$ and a regular path query r, all pairs $(n, n') \in V \times V$ of nodes such that G contains a path $n \rightsquigarrow_p n'$ such that $l(p) \in L(r)$.

3. The Algorithm

In this section, we first give the overview of the algorithm and some related notions, then present the details of the algorithm.

3.1 Overview

Let r be a regular path query and $G = (V, E, \Sigma)$ be a graph. The algorithm uses an area S allocated in main memory with $|S| = \epsilon \cdot M$, where $|S|$ denotes the size of S, M is the size of main memory, and $0 < \epsilon < 1$. The input data to our algorithm is a node list N (Fig. 1 (A)), which consists of the nodes of V with some information of each node (outgoing edges, etc.). In short, the algorithm works as follows.

1. Firstly, the algorithm reads N sequentially and repeats the following until EOF is found.
   a. Load $|S|$ bytes of data from N into S. Thus S contains a subgraph of G.
   b. Traverse S and do the following.
      i. Find the "local" answers (paths matching r) in S that can be obtained by traversing only S (Fig. 1 (i)).
      ii. To find answer paths running across boundaries of S, find paths adjacent to edges outside S, and store their connections into another graph file called *contracted graph* (Fig. 1 (ii) and (B)).
2. Then, the algorithm traverses the contracted graph obtained in step (b-ii) and outputs the answers not found in step (b-i) (Fig. 1 (iii)).

Note that, in this and the subsequent examples, for simple and clear explanation we use a DFA as an example.

Let us give some definitions related to node list N. Let $n \in V$ be a node. By $order(n)$ we mean the *order* of n in N. For example, in Fig. 1 (A) $order(n_8) = 1$, $order(n_2) = 2$, and so on. By $S.min$ we mean the minimum order of the nodes in S. Similarly, by $S.max$ we mean the maximum order of the nodes in S. For example, assuming that the second region of N in Fig. 1 (A) is loaded into S, we have $S.min = order(n_5) = 5$ and $S.max = order(n_1) = 8$. By $Out(n)$ we mean the set of outgoing edges of n, that is, $Out(n) = \{n \xrightarrow{l} n' \mid n' \in V, n \xrightarrow{l} n' \in E\}$. By $In(n, l)$ we mean the set of source nodes of incoming edges of n labeled by l, that is, $In(n, l) = \{n' \in V \mid n' \xrightarrow{l} n \in E\}$. By $inMax(n, l)$, we mean the node having the maximum order in $In(n, l)$, that is,

$$inMax(n, l) = \operatorname*{argmax}_{n' \in In(n,l)} order(n').$$

Similarly, we define that

$$inMin(n, l) = \operatorname*{argmin}_{n' \in In(n,l)} order(n').$$

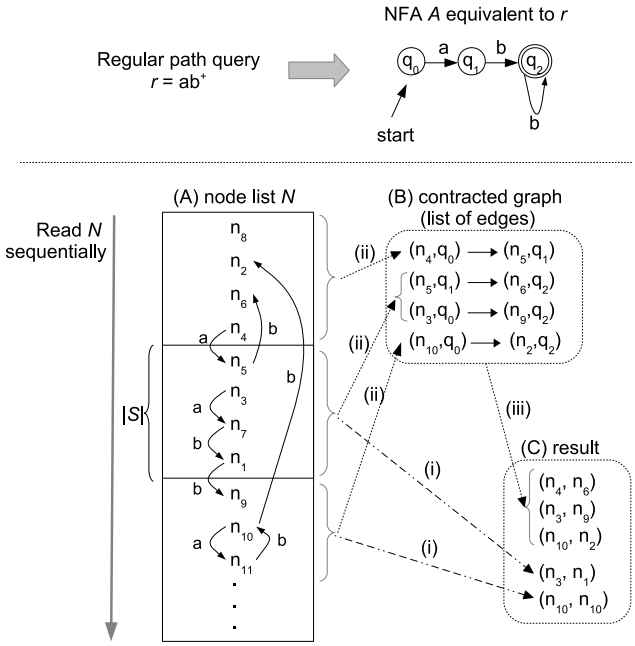


Fig. 1 Outline of our algorithm

Let r be a regular path query and $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA such that $L(A) = L(r)$. Suppose that $|S|$ bytes of data is loaded from N into S. If S contains a path $n \rightsquigarrow_p n'$ such that $l(p) \in L(A)$, then the algorithm outputs (n, n'). For example, in Fig. 1 (A) the algorithm outputs (n_3, n_1) as an answer due to $n_3 \xrightarrow{a} n_7 \xrightarrow{b} n_1$, which can be obtained by traversing only S. However, S contains only a subgraph of G, and thus we also have to handle edges running across a boundary of S appropriately, i.e., (i) edges from outside to inside S (e.g., $n_4 \xrightarrow{a} n_5$ in Fig. 1 (A)) and (ii) edges from inside to outside S (e.g., $n_1 \xrightarrow{b} n_9$ in Fig. 1 (A)). We have to find paths matching r even if the paths contain such "boundary" edges.

To find such paths, our algorithm traverses S and finds paths $n \rightsquigarrow_p n'$ such that $q' \in \hat{\delta}(q, l(p))$ for some $q, q' \in Q$ (i.e., p matches a subexpression of r), then check if p is adjacent to a boundary edge mentioned above (i) and (ii). To do this, for a pair (n, q) of a node n in S and a state $q \in Q$, we define four types T_{in}, T_{out}, T_{start}, and T_{accept}, as follows. We write $T_{in}(n, q)$ if a pair (n, q) is of type T_{in} (T_{out}, T_{start}, T_{accept} are denoted similarly).

- $T_{in}(n, q)$ iff n has an incoming edge that enters S and matches a state transition of A, that is, for some $l \in \Sigma$,

  - $q \in \delta(q', l)$ for some $q' \in Q$, and
  - n has an incoming edge labeled by l entering S, that is, $order(inMin(n, l)) < S.min$ or $order(inMax(n, l)) > S.max$.

- $T_{out}(n, q)$ iff n has an outgoing edge that leaves S and that matches a state transition of A, that is, for some $n \xrightarrow{l} n' \in Out(n)$,

  - $q' \in \delta(q, l)$ for some $q' \in Q$, and
  - n' is a node outside S, that is, $order(n') < S.min$ or $order(n') > S.max$.

  For the node n' and the state q' above, we say that (n', q') is a *certificate* of $T_{out}(n, q)$.
- $T_{start}(n, q)$ iff $q = q_0$.
- $T_{accept}(n, q)$ iff $q \in F$.

T_{in} and T_{out} are used to check if (n, q) is adjacent to a "boundary" edge, while T_{start} and T_{accept} are used to check if (n, q) is a pair at which a traversal should be started/finished.

With the four types we formally define contracted graph. We assume that a state is represented by an integer id. For pairs $(n, q), (n', q')$ of nodes and states, we write $(order(n), q) < (order(n'), q')$ if $order(n) < order(n)$ or $order(n) = order(n)$ and $q < q'$. The algorithm "partitions" node list N into $\lceil \frac{|N|}{|S|} \rceil$ sublists and loads each sublist into S one by one. An edge $n \xrightarrow{l} n'$ is called a *boundary edge* if n and n' are in distinct sublists. Then the *contracted graph* of G, denoted *cgraph*, is a list of unlabeled directed edges satisfying the following two conditions.

1. An edge $(n, q) \to (n', q')$ is in *cgraph* iff for the sublist N' of N containing n, one of the following conditions holds.
   - For some $n'' \in V$ and some $q'' \in Q$, $T_{start}(n, q)$, $T_{out}(n'', q'')$, N' contains a path $n \rightsquigarrow_p n''$ such that $q'' \in \hat{\delta}(q, l(p))$, and (n', q') is a certificate of $T_{out}(n'', q'')$.
   - For some $n'' \in V$ and some $q'' \in Q$, $T_{in}(n, q)$, $T_{out}(n'', q'')$, N' contains a path $n \rightsquigarrow_p n''$ such that $q'' \in \hat{\delta}(q, l(p))$, and (n', q') is a certificate of $T_{out}(n'', q'')$.
   - $T_{in}(n, q)$, $T_{accept}(n', q')$, and N' contains a path $n \rightsquigarrow_p n'$ such that $q' \in \hat{\delta}(q, l(p))$.
2. The edges in *cgraph* are listed in ascending order of the heads of the edges (see Fig. 1 (B)). That is, $(n_1, q_1) \to (n_2, q_2)$ precedes $(n_3, q_3) \to (n_4, q_4)$ in *cgraph* whenever $(order(n_1), q_1) < (order(n_3), q_3)$.

The idea of T_{in} and T_{out} is analogous to the notion of input/output markers used in the UnQL data model [30]. In the data model, input/output markers are used to describe structural recursion on graphs containing cycles. Since structural recursion cannot be directly applied to a graph containing cycles, in the data model a graph is firstly split into small cycle-free pieces, then structural recursion is applied on the pieces and the results are glued together. Input/output markers are used to glue such pieces; a node with an output marker in some piece is glued with a node with an input marker in another piece. Similarly, in our algorithm, a node of type T_{out} in a subgraph is "glued" with a node of type T_{in} in another subgraph.

3.2 Details of the Algorithm

We present the details of our algorithm. Let $G = (V, E, \Sigma)$ be a graph and r be a regular path query. First of all, we define node list N: N is a list consisting of the nodes in V with the following items for each node $n \in V$; $Out(n)$, $inMin(n)$, $inMax(n)$, and $order(n)$.

Let us first present the main part of our algorithm. This procedure reads N sequentially and processes each subgraph loaded into S. For each loaded subgraph, it deletes unnecessary edges that need not to be visited during traversing the subgraph (line 7), and then finds the answers obtained by traversing S and constructs the contracted graph *cgraph* simultaneously (line 8). After the sequential scan of N, the algorithm finds the answers obtained by traversing *cgraph* (line 10).

Procedure Main
Input: Node list N of graph $G = (V, E, \Sigma)$, regular path query r
Output: Pairs of nodes (n, n') such that $n \rightsquigarrow_p n'$ and that $l(p) \in L(r)$

1. Create an empty file *cgraph*
2. Construct an NFA $A = (\Sigma, Q, \delta, q_0, F)$ such that $L(A) = L(r)$
3. Allocate an area S in main memory ($|S| = \epsilon \cdot M$)
4. Read N sequentially and do the following until EOF is found
5. **begin**
6. Load $|S|$ bytes of data from N into S
7. Prune(S, A)
8. TraverseBuf(S, A, *cgraph*)
9. **end**
10. TraverseCgraph(A, *cgraph*)

In the following, we present the details of TraverseBuf (line 8), TraverseCgraph (line 10), and Prune (line 7).

3.2.1 Processing Node List

We next present TraverseBuf, which finds the "local" answers in S and constructs *cgraph* simultaneously. Let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA such that $L(A) = L(r)$. This procedure works as follows. For each pair (n, q) of a node n in S and a state q, if $T_{in}(n, q)$ or $T_{start}(n, q)$, then we start a traversal from (n, q)[†] (lines 2 to 4). Let (n', q') be a pair encountered in this traversal. If $T_{start}(n, q)$ and $T_{accept}(n', q')$, then (n, n') is outputted as an answer (lines 6 and 7). If $T_{in}(n, q)$ and $T_{accept}(n', q')$, then we have "(n', q') is reachable from (n, q)". To record this, $(n, q) \to (n', q')$ is added to *cgraph* (lines 8, 9, and 13). Consider the case where $T_{out}(n', q')$

[†]Our current implementation adopts breadth first search but any other search strategies can be applicable. Finding the best search strategy is left as a future work.

(line 10). By the definition of T_{out}, there exists a certificate (n'', q'') of $T_{out}(n', q')$, in other words, "(n'', q'') is reachable from (n, q)". To record this, for each certificate (n'', q'') of $T_{out}(n', q')$, $(n, q) \to (n'', q'')$ is added to *cgraph* (lines 10 to 13).

Procedure TraverseBuf

Input: Subsequence of N loaded into S, NFA $A = (Q, \Sigma, \delta, q_0, F)$, contracted graph *cgraph* (empty file)
Output: Pairs (n, n') of nodes in S such that $n \rightsquigarrow_p n'$ and that $l(p) \in L(r)$, contracted graph *cgraph*

1. $tmp \leftarrow \emptyset$
2. **for each** pair (n, q) of a node n in S and $q \in Q$ **do**
3. **if** $T_{in}(n, q)$ or $T_{start}(n, q)$ **then**
4. Traverse S and compute the following set I.
$I = \{(n', q') \mid n \rightsquigarrow_p n', q' \in \hat{\delta}(q, l(p)),$
$T_{out}(n', q')$ or $T_{accept}(n', q')\}$.
5. **for each** $(n', q') \in I$ **do**
6. **if** $T_{start}(n, q)$ and $T_{accept}(n', q')$ **then**
7. Output (n, n') as an answer
8. **if** $T_{in}(n, q)$ and $T_{accept}(n', q')$ **then**
9. $tmp \leftarrow tmp \cup \{(n, q) \to (n', q')\}$
10. **if** $T_{out}(n', q')$ **then**
11. **for each** certificate (n'', q'') of $T_{out}(n', q')$ **do**
12. $tmp \leftarrow tmp \cup \{(n, q) \to (n'', q'')\}$
13. Add each edge in *tmp* to *cgraph* in ascending order

In line 13, the edges $(n, q) \to (n', q')$ in *tmp* are added to *cgraph* in ascending order of pair $(order(n), q)$ according to the condition 2 of the definition of *cgraph*.

3.2.2 Processing Contracted Graph

Then we present TraverseCgraph, which traverses *cgraph* and outputs the rest of answers. If the size of *cgraph* does not exceed $|S|$, then it suffice to load entire *cgraph* into S and find the answers by traversing S (lines 1 to 4). In line 3, we write $(n, q_0) \rightsquigarrow (n', q')$ if there exists a path from (n, q_0) to (n', q') in S. In most cases, *cgraph* is small and the traversal of *cgraph* is completed here. On the other hand, if the size of *cgraph* exceeds $|S|$, the procedure repeats a forward scan of *cgraph* (line 8) and a backward scan of *cgraph* (line 9) alternatively, until all the answers contained in *cgraph* are found.

Procedure TraverseCgraph

Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$, contracted graph *cgraph*
Output: Pairs (n, n') of nodes such that $(n, q_0) \rightsquigarrow (n', q')$ for some $q' \in F$

1. **if** the size of *cgraph* does not exceed $|S|$ **then**
2. Load entire *cgraph* into S
3. Traverse S and find all pairs (n, n') such that $(n, q_0) \rightsquigarrow (n', q')$ and that $q' \in F$
4. Output the pairs obtained in line 3
5. **else**
6. Create priority queues pq_fwd and pq_bwd
Initially, pq_fwd and pq_bwd are empty
7. **do**
8. ScanForward(A, *cgraph*, pq_fwd, pq_bwd)
9. ScanBackward(A, *cgraph*, pq_fwd, pq_bwd)
10. **while** $pq\_fwd \neq \emptyset$ or $pq\_bwd \neq \emptyset$

In the following, we present ScanForward and ScanBackward in lines 8 and 9 above. To describe these procedures we need some notations. Suppose that a subpart of *cgraph* is loaded into S, and let $(n_f, q_f) \to (n'_f, q'_f)$ be the first edge in S and $(n_l, q_l) \to (n'_l, q'_l)$ be the last edge in S. Then we define that $S.min' = (order(n_f), q_f)$ and that $S.max' = (order(n_l), q_l)$. Note that, by the ordering of *cgraph* (line 13 of TraverseBuf), for any edge $(n, q) \to (n', q')$ in S we have $S.min' \leq (order(n), q) \leq S.max'$.

ScanForward and ScanBackward use two priority queues pq_fwd and pq_bwd. Suppose that, during traversing S, we encounter an edge $(n, q) \to (n', q')$ such that (n', q') is outside S, i.e., $(order(n'), q') < S.min'$ or $(order(n'), q') > S.max'$. Since (n', q') is outside S, we have to "suspend" the traverse and "restart" it from (n', q') afterwards. pq_fwd and pq_bwd are used to remember such pairs for suspending and restarting. pq_fwd and pq_bwd are defined as follows.

- pq_fwd is an ascending priority queue w.r.t. $(order(n), q)$, thus pair (n, q) with the least $(order(n), q)$ is at the top of the queue.
- pq_bwd is a descending priority queue w.r.t. $(order(n), q)$, thus pair (n, q) with the largest $(order(n), q)$ is at the top of the queue.

Now let us present ScanForward (ScanBackward is defined similarly). The procedure reads *cgraph* sequentially and repeats the following until EOF is found. First, the procedure loads $|S|$ bytes of data from *cgraph* into S, and computes set I of pairs at which a traversal should be (re)started. At the first execution of ScanForward, for any edge $(n, q) \to (n', q')$ in S, $(n, q)_{(n)}$ is added to I whenever $q = q_0$ in lines 5 and 6. Here, $(n, q)_{(n)}$ is called a *subscripted pair*, and the subscript (n) of $(n, q)_{(n)}$ records the node at which the traversal is originally started. As we will see below, the subscript is used when outputting an answer. In line 7, the "visited" mark is to prevent a subscripted pair from being visited more than once. At the second or later execution, any pair $(n, q)_{(n_s)}$ in pq_fwd is added to I if a traversal can be restarted from (n, q) on the current sublist in S, i.e., $S.min' \leq (order(n), q) \leq S.max'$ and S contains an edge $(n, q) \to (n', q')$ for some (n', q') (line 8). Then for each subscripted pair $(n, q)_{(n_s)}$ in I, a traversal on S from (n, q) is (re)started (line 10). Let (n', q') be a pair encountered by this traversal. If $q' \in F$, then (n_s, n') is outputted as an answer, where n_s is the subscript of the subscripted pair $(n, q)_{(n_s)}$ at which the traversal is (re)started (lines 13 and 14). If (n', q') is a pair outside S, then the traversal is suspended and $(n', q')_{(n_s)}$ is added to pq_fwd or pq_bwd (lines 15 to 18).

Procedure ScanForward

Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$, contracted graph *cgraph*, priority queues pq_fwd, pq_bwd
Output: Pairs (n, n') such that $(n, q_0) \rightsquigarrow (n', q')$ and that $q' \in F$ obtained by scanning *cgraph* forward

1. Do the following until EOF is found
2. **begin**

```
3.    I ← ∅
4.    Load |S| bytes of data from cgraph
5.    if this is the first execution of ScanForward then
6.        I ← {(n,q)(n) | (n,q) → (n′,q′) is an edge in S,
          q = q0}
7.        Mark (n,q)(n) as "visited" for each (n,q)(n) ∈ I
8.    Remove all pairs (n,q)(ns) such that
      S.min′ ≤ (order(n),q) ≤ S.max′ from pq_fwd.
      For each removed pair (n,q)(ns), add (n,q)(ns) to I if
      S contains an edge (n,q) → (n′,q′) for some (n′,q′)
9.    for each (n,q)(ns) ∈ I do
10.       Traverse S from (n,q) and find all pairs (n′,q′)
          such that (n,q) ⇝ (n′,q′) in S and that
          (n′,q′)(ns) is unvisited.
          Let T be the resulting set.
11.       Mark (n′,q′)(ns) as "visited" for each (n′,q′)(ns) ∈ T
12.       for each (n′,q′)(ns) ∈ T do
13.           if q′ ∈ F then
14.               Output (ns,n′) as an answer
15.           if (order(n′),q′) < S.min′ then
16.               Add (n′,q′)(ns) to pq_bwd
17.           else if (order(n′),q′) > S.max′ then
18.               Add (n′,q′)(ns) to pq_fwd
19. end
```

Let us explain ScanForward and ScanBackward by examples. According to TraverseCgraph, ScanForward is executed first, then ScanBackward is executed, and so on. Firstly, for any edge $(n,q) \to (n',q')$ in S, ScanForward starts a traversal from (n,q) if $q = q_0$ (lines 6 and 9). For example, consider an edge $(n_{12},q_0) \to (n_{10},q_1)$ in Fig. 2 (A). Since the state q_0 of (n_{12},q_0) is the start state, we start a traversal from this edge. However, as shown in Fig. 2 (A), $(n_{10},q_1) \to (n_2,q_2)$ is outside S (in a backward direction), i.e., $(order(n_{10}),q_1) < S.min'$. Thus we have to "suspend" the traversal and "restart" it from (n_{10},q_1) later. To record this, we add $(n_{10},q_1)_{(n_{12})}$ to pq_bwd in lines 15 and 16. As an another example, consider an edge $(n_9,q_0) \to (n_8,q_1)$ in S (Fig. 2 (A)). The edge is adjacent to $(n_8,q_1) \to (n_{11},q_2)$, which is outside S (in a forward direction), i.e., $(order(n_8),q_1) > S.max'$. Thus $(n_8,q_1)_{(n_9)}$ is added to pq_fwd (lines 17 and 18). We use pq_fwd and pq_bwd separately according to scanning directions. Since pq_fwd is an ascending priority queue, ScanForward looks pq_fwd to obtain pairs at which traversals should be restarted in line 8 (i.e., the scanning direction coincides with the node ordering of pq_fwd). On the other hand, pq_bwd is a descending priority queue, and thus ScanBackward looks pq_bwd to obtain pairs at which traversals should be restarted.

After the first execution of ScanForward, ScanBackward is executed in which *cgraph* is scanned in a backward direction (Fig. 2 (B)). Here, for each edge $(n,q) \to (n',q')$ in S, if $(n,q)_{(n_s)}$ is contained in pq_bwd, then we "restart" the traversal from (n,q). And if a pair (n'',q'') with $q'' \in F$ is found during the traversal, then this is outputted as an answer. For example, consider an edge $(n_{10},q_1) \to (n_2,q_2)$ in Fig. 2 (B). Then $(n_{10},q_1)_{(n_{12})}$ is contained in pq_bwd due to the previous ScanForward execution, and thus the traversal is restarted. We repeat such scans until both pq_fwd and pq_bwd become empty.

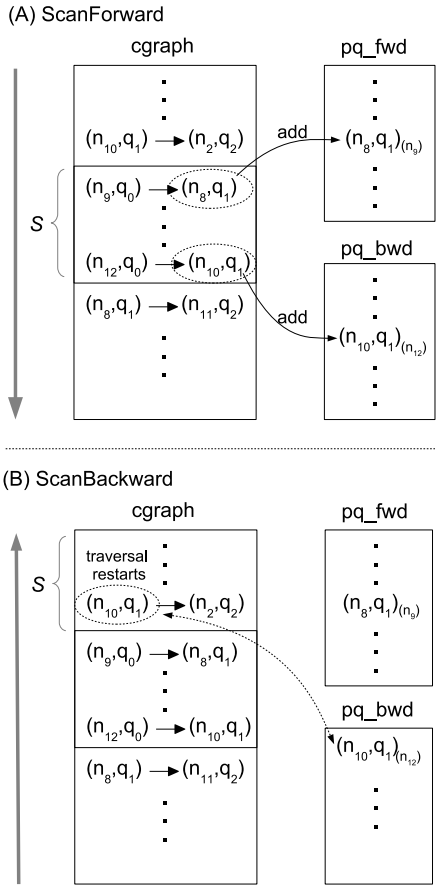


Fig. 2 ScanForward and ScanBackward

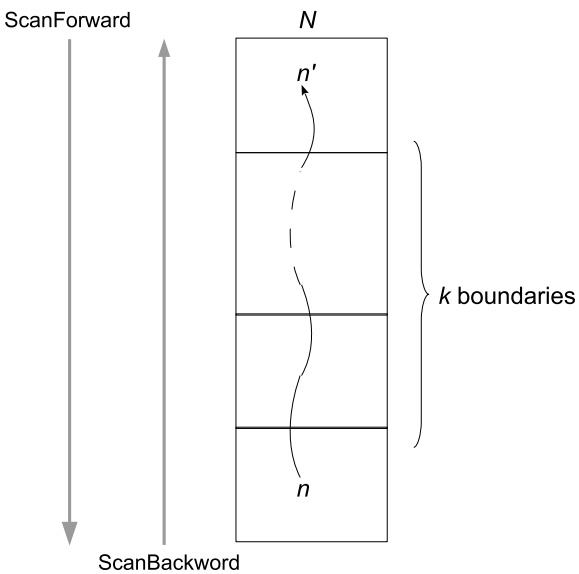


Fig. 3 A path from n to n'

Actually, *cgraph* can be traversed by ScanForward only, although the algorithm uses both ScanForward and ScanBackward. The reason why we use ScanBackward as well as ScanForward is that "upward" paths can be traversed more efficiently by using ScanBackward. For example, consider the path from n to n' in Fig. 3, which runs

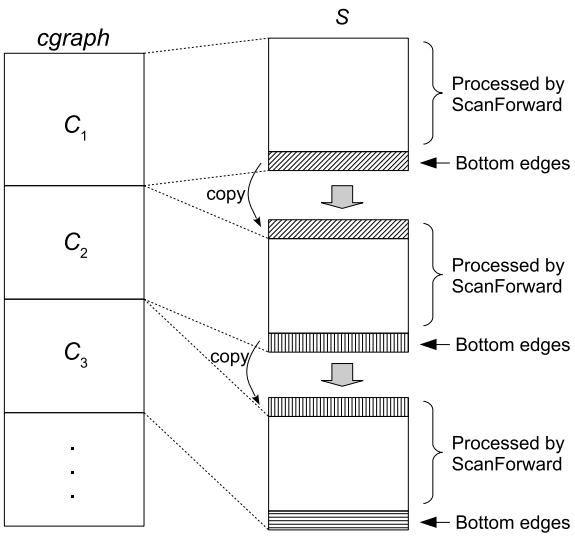


Fig. 4 Loading sublists of *cgraph* into S

across k boundaries. Then only one ScanBackward is required to traverse the path. On the other hand, without ScanBackward we need k ScanForwards' to traverse the path.

Finally, we note that a little care needs to be taken when loading a sublist into S in line 4 of ScanForward. If *cgraph* contains more than one edge having the same head, say $e_1 = (n_1, q_1) \to (n_2, q_2)$ and $e_2 = (n_1, q_1) \to (n_3, q_3)$, then e_1 and e_2 may be put into distinct sublists. If this happens, however, the "restart" of a traversal does not work appropriately. For example, if we find a pair $(n_1, q_1)_{(n)}$ in a priority queue during processing the sublist containing e_2, then a traversal is restarted at e_2 but it fails to visit e_1 since e_1 is not contained in the sublist. To cope with this problem, sublists of *cgraph* are loaded into S in the following manner. We say that an edge e is a *bottom edge* of S if e is the last edge in S or the head of e coincides with that of the last edge of S. As shown in Fig. 4, when C_1 is loaded into S, ScanForward processes the edges in S *except* the bottom edges. When it is completed, the bottom edges are copied to the top of S, and then C_2 is loaded into S just below the copied edges. Then ScanForward processes the edges in S *except* the bottom edges, and so on.

3.2.3 Pruning Edges

In line 7 of Main, Prune(S, A) deletes edges unnecessary w.r.t. NFA A from S. Let n be a node. This procedure determines (i) which outgoing edges of n can be pruned according to the transition function of A and the incoming edges of n and (ii) which incoming edges of n can be pruned according to the transition function of A and the outgoing edges of n. For simplicity, we explain this procedure by an example. Consider pruning outgoing edges of n_1 in Fig. 5 (b). It is clear that $n_1 \xrightarrow{f} n_7$ can be pruned since f does not appear in NFA A in Fig. 5 (a). On the other hand, $n_1 \xrightarrow{a} n_4$ is not pruned since $\delta(q_0, a) \neq \emptyset$ and thus this edge is required for a traversal starting from (n_1, q_0) (lines 2 and 3 of TraverseBuf). $n_1 \xrightarrow{b} n_5$ is not pruned either since we have $q_2 \in \hat{\delta}(q_0, ab)$ and $n_2 \xrightarrow{a} n_1 \xrightarrow{b} n_5$, i.e., (n_5, q_2) is reachable from (n_2, q_0). Finally, $n_1 \xrightarrow{c} n_6$ is pruned since the incoming edge $n_2 \xrightarrow{a} n_1$ is labeled by a but $\hat{\delta}(q_0, ac) = \emptyset$.

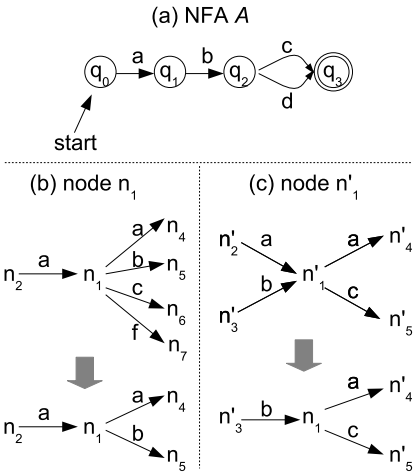


Fig. 5 Pruning outgoing/incoming edges

Incoming edges are pruned similarly. Consider the incoming edges of n'_1 in Fig. 5 (c). First, $n'_3 \xrightarrow{b} n'_1$ is not pruned since n'_1 has an outgoing edge $n'_1 \xrightarrow{c} n'_5$ and $q_3 \in \hat{\delta}(q_1, bc)$, i.e., (n'_5, q_3) is reachable from (n'_3, q_1). Then consider $n'_2 \xrightarrow{a} n'_1$. (n'_1, q_1) is reachable from (n'_2, q_0) and we have $\delta(q_1, b) = \{q_2\}$, but n'_1 has no outgoing edge labeled by b (any traversal is stopped here). Thus $n'_2 \xrightarrow{a} n'_1$ is redundant and it is pruned.

4. Correctness of the Algorithm

In this section, we show the correctness of the algorithm. We first show that the edges of *cgraph* are obtained correctly (Lemma 1). Then we show that *cgraph* is traversed correctly (Lemma 2). By using these lemmas we show the correctness of the algorithm (Theorem 1).

Let $G = (V, E, \Sigma)$ be a graph, r be a regular path query, and $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA such that $L(A) = L(r)$. We say that (n', q') is *reachable* from (n, q) in G (w.r.t. A) if there is a path $n \rightsquigarrow_p n'$ in G such that $q' \in \hat{\delta}(q, l(p))$. In particular, if the path p contains no boundary edge, then we say that (n', q') is 0-*reachable* from (n, q) in G.

Lemma 1: If (n', q') is 0-reachable from (n, q) in G, then the following four conditions hold.

1. If $T_{start}(n, q)$ and $T_{out}(n', q')$, then for any certificate (n'', q'') of $T_{out}(n', q')$, *cgraph* contains an edge $(n, q) \to (n'', q'')$.
2. If $T_{start}(n, q)$ and $T_{accept}(n', q')$, then (n, n') is outputted as an answer.
3. If $T_{in}(n, q)$ and $T_{out}(n', q')$, then for any certificate (n'', q'') of $T_{out}(n', q')$, *cgraph* contains an edge

$(n, q) \to (n'', q'')$.

4. If $T_{in}(n, q)$ and $T_{accept}(n', q')$, then *cgraph* contains an edge $(n, q) \to (n', q')$.

Proof: Assume that (n', q') is 0-reachable from (n, q) in G. Then there is a path $n \rightsquigarrow_p n'$ in G such that $q' \in \hat{\delta}(q, l(p))$ and that p contains no boundary edge. Since p contains no boundary edge, TraverseBuf finds p and adds appropriate edges to *cgraph* and/or outputs answers according the antecedents of the above conditions 1 to 4. □

We next show that *cgraph* is correctly traversed by TraverseCgraph. If there is a path from (n, q) to (n', q') in *cgraph*, then we say that (n, q) is *c-reachable* from (n', q').

Lemma 2: (n', q') is c-reachable from (n, q_0) for some $q' \in F$ iff TraverseCgraph outputs (n, n') as an answer.

Proof: If the size of *cgraph* does not exceed $|S|$, the lemma follows from lines 1 to 4 of TraverseCgraph. Consider the case where the size of *cgraph* exceeds $|S|$. In the following, we consider the sufficient condition of the lemma. TraverseCgraph partitions *cgraph* into $\left\lceil \frac{|cgraph|}{|S|} \right\rceil$ sublists. For two edges $(n, q) \to (n', q')$ and $(n', q') \to (n'', q'')$, if the edges are in distinct sublists, then we say that $(n, q) \to (n', q')$ is a *boundary edge*. Let p be a path containing a boundary edge. Then p can be denoted $(n, q) \rightsquigarrow (n', q') \to (n'', q'') \rightsquigarrow (n''', q''')$, where $(n', q') \to (n'', q'')$ is a boundary edge. We say that (n'', q'') is a *boundary pair* of p. For a path p in *cgraph*, by $bp(p)$ we mean the number of boundary pairs in p.

Suppose that (n', q') is c-reachable from (n, q_0). Then there is a path $(n, q_0) \rightsquigarrow_p (n', q')$ in *cgraph*. Here, suppose that a boundary pair, say (n'', q''), occurs in p twice. Then p can be denoted as follows.

$$(n, q_0) \rightsquigarrow (n'', q'') \rightsquigarrow (n'', q'') \rightsquigarrow (n', q').$$

By skipping the subpath between the boundary pairs, we obtain

$$(n, q_0) \rightsquigarrow (n'', q'') \rightsquigarrow (n', q').$$

Therefore, if (n', q') is c-reachable from (n, q_0), then there is a path from (n, q_0) to (n', q') such that no boundary pair occurs more than once. For a path p, if no boundary pair occurs more than once in p, then we say that p is *b-simple*. Note that for any b-simple path p, $bp(p) \le |cgraph|$.

In the following, we show that if (n', q') is c-reachable from (n, q_0) with $q' \in F$, then the algorithm finds a b-simple path from (n, q_0) to (n', q'). We need a few definitions. We write $(n, q) \rightsquigarrow_k^b (n', q')$ if (i) $k = 0$ and $(n, q) = (n', q')$ or (ii) $k \ge 1$ and there is a b-simple path $(n, q) \rightsquigarrow_p (n', q')$ such that $bp(p) \le k$ and that (n', q') is a boundary pair. Similarly, we write $(n, q) \rightsquigarrow_k^a (n', q')$ if there is a b-simple path $(n, q) \rightsquigarrow_p (n', q')$ such that $bp(p) \le k$ and that $q' \in F$.

We show by induction on k that the following conditions hold.

1. If $(n, q_0) \rightsquigarrow_k^b (n', q')$, then $(n', q')_{(n)}$ is added to *pq_fwd* or *pq_bwd*.
2. If $(n, q_0) \rightsquigarrow_k^a (n', q')$, then (n, n') is outputted as an answer.

Basis: Let $k = 0$. Consider the condition 1. By lines 5 and 6 of ScanForward, for every edge $(n, q) \to (n'', q'')$, $(n, q)_{(n)}$ is added to *pq_fwd* whenever $q = q_0$. As for the condition 2, $(n, q_0) \rightsquigarrow_0^a (n', q')$ implies that there is a path $(n, q_0) \rightsquigarrow_p (n', q')$ such that $bp(p) = 0$ and that $q' \in F$. Since $bp(p) = 0$, (n, n') is outputted as an answer by lines 13 and 14 of ScanForward.

Induction: Consider the condition 1 (the condition 2 can be shown similarly). Assume as the induction hypothesis that if $(n, q_0) \rightsquigarrow_{k-1}^b (n', q')$, then the condition 1 holds. Suppose that $(n, q_0) \rightsquigarrow_k^b (n', q')$. Then there is a b-simple path $(n, q_0) \rightsquigarrow_p (n', q')$ such that $bp(p) \le k$ and that (n', q') is a boundary pair. Then p can be denoted

$$\underbrace{(n, q_0) \rightsquigarrow \overbrace{(n'', q'')}^{p_2}}_{p_1} \rightsquigarrow (n''', q''') \to (n', q'),$$

where (n'', q'') and (n', q') are boundary pairs, $(n''', q''') \to (n', q')$ is a boundary edge, $bp(p_1) \le k - 1$, and $bp(p_2) = 1$ (i.e., p_2 contains no boundary pair except (n'', q'')). Since $bp(p_1) \le k - 1$ and (n'', q'') is a boundary pair, we have $(n, q_0) \rightsquigarrow_{k-1}^b (n'', q'')$. Thus by the induction hypothesis $(n'', q'')_{(n)}$ is added to *pq_fwd*/*pq_bwd*. Since $(n'', q'')_{(n)}$ can be found in *pq_fwd*/*pq_bwd*, the traversal of p is restarted from (n'', q''). Moreover, since p_2 contains no boundary pair except (n'', q'') and (n', q') is a boundary pair occurring in neither p_1 nor p_2, $(n', q')_{(n)}$ is added to *pq_fwd*/*pq_bwd* by lines 15 to 18 of ScanForward. □

We now have the following theorem.

Theorem 1: Let $n, n' \in V$ be nodes and $q' \in F$ be an accepting state. Then (n', q') is reachable from (n, q_0) in G iff the algorithm outputs (n, n').

Proof: The necessary condition can be shown easily. In the following, we consider the sufficient condition. If (n', q') is 0-reachable from (n, q_0) in G, then (n, n') is outputted as an answer by the condition 2 of Lemma 1. Consider the case where (n', q') is reachable from (n, q_0) but not 0-reachable. Then there is a path $n \rightsquigarrow_p n'$ in G such that $q' \in \hat{\delta}(q_0, l(p))$ and that p contains at least one boundary edge. Let k be the number of boundary edges in p. Then p can be denoted

$$n \rightsquigarrow_{p_1} n_1 \xrightarrow{l_1} n_2 \rightsquigarrow_{p_2} n_3 \xrightarrow{l_2} n_4 \rightsquigarrow_{p_3} \cdots \xrightarrow{l_k} n_{2k} \rightsquigarrow_{p_{k+1}} n',$$

where p_i is a path containing no boundary edge ($1 \le i \le k + 1$) and $n_{2i-1} \xrightarrow{l_i} n_{2i}$ is a boundary edge ($1 \le i \le k$). Then since $q' \in \hat{\delta}(q_0, l(p))$ and $q' \in F$, for some $q_1, q_2, \cdots, q_{2k} \in Q$ we have the following.

P_1: p_1 satisfies that $T_{start}(n, q_0)$, $q_1 \in \hat{\delta}(q_0, l(p_1))$, and that $T_{out}(n_1, q_1)$, and $n_1 \xrightarrow{l_1} n_2$ is a boundary edge with $q_2 \in \delta(q_1, l_1)$.

P_i: For every $2 \le i \le k$, p_i satisfies that $T_{in}(n_{2i-2}, q_{2i-2})$, $q_{2i-1} \in \hat{\delta}(q_{2i-2}, l(p_i))$, and that $T_{out}(n_{2i-1}, q_{2i-1})$, and

$n_{2i-1} \xrightarrow{l_i} n_{2i}$ is a boundary edge with $q_{2i} \in \delta(q_{2i-1}, l_i)$.

P_{k+1}: p_{k+1} satisfies that $T_{in}(n_{2k}, q_{2k})$, $q' \in \hat{\delta}(q_{2k}, l(p_{k+1}))$, and that $T_{accept}(n', q')$.

By Lemma 1, any edges corresponding to P_1, P_i, P_{k+1} are contained in *cgraph*, and *cgraph* is processed correctly by Lemma 2. Hence the theorem holds. □

5. I/O and CPU Costs of the Algorithm

In this section, we briefly present the I/O and CPU costs of the algorithm (more details can be found in the appendix). Let $G = (V, E, \Sigma)$ be a graph, r be a regular path query, and $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA such that $L(A) = L(r)$.

5.1 I/O Cost

Following External Memory Model [31], we assume that data is transferred between HDD and main memory in blocks of size B.

Since node list N contains $Out(n)$, $order(n)$, and $inMax(n, l)$ and $inMin(n, l)$ for each node n, the size of N is in $O(|E| + |V|)$. Thus, the I/O cost of reading N sequentially is in $O((|E|+|V|)/B)$. Moreover, the cost of writing *cgraph* is in $O(|cgraph|/B)$. Suppose that the size of *cgraph* does not exceed $|S|$. Then TraverseCgraph reads *cgraph* only once, where $|cgraph| < |S| < |N|$. Let Ans be the set of answers. Taking the cost of outputting Ans into account, the total I/O cost of Main is

$$O((|E| + |V| + output(|Ans|))/B). \quad (1)$$

Here, $|Ans|$ is in $O(|V_s(r)||V_a(r)|)$, where $V_s(r)$ is the set of "possible start nodes" and $V_a(r)$ is the set of "possible accepting nodes", that is,

$$V_s(r) = \{n \in V \mid n \xrightarrow{l} n' \in E, \delta(q_0, l) \neq \emptyset\},$$
$$V_a(r) = \{n' \in V \mid n \xrightarrow{l} n' \in E, \delta(q, l) \cap F \neq \emptyset \text{ for some } q \in Q\}.$$

Suppose next that the size of *cgraph* exceeds $|S|$. Then we can show that TraverseCgraph requires $O(|cgraph|^2/B)$ I/O read cost. Thus the total I/O cost of Main is

$$O((|E|+|V|+|cgraph|^2+output(|Ans|))/B). \quad (2)$$

Here, $|cgraph|$ is in

$$O(|V_{in}(r)||E(r)||\Delta|), \quad (3)$$

where $V_{in}(r) = \{n' \in V \mid n \xrightarrow{l} n' \in E(r)\}$, $E(r) = \{n \xrightarrow{l} n' \in E \mid l \text{ appears in } r\}$, and $\Delta = \{(q, q') \in Q \times Q \mid q' \in \hat{\delta}(q, w), w \in \Sigma^*\}$.

This suggests that *cgraph* does not become too large whenever Prune deletes enough number of unnecessary edges.

5.2 CPU Cost

Since N is partitioned into $k = \lceil \frac{|N|}{|S|} \rceil$ sublists, lines 5 to 9 of Main are repeated k times. For each sublist, since the number of edges in S is in $O(|E|/k)$, Prune in line 7 can be done in $O(|E||\Delta|/k)$ time. Consider TraverseBuf in line 8 of Main. Let

$$V_{out}(r) = \{n \in V \mid n \xrightarrow{l} n' \in E(r)\}.$$

Then we can show that TraverseBuf processes each sublist in $O(|V_{out}(r)||E(r)||Q||\Delta|/k)$ time. Thus the cost of Main except TraverseCgraph is in

$$O(k(|E||\Delta|/k + |V_{out}(r)||E(r)||Q||\Delta|/k)) = O(|E||\Delta| + |V_{out}(r)||E(r)||Q||\Delta|).$$

Consider next the cost of TraverseCgraph. The numbers of nodes and edges in *cgraph* are in $O(|cgraph|)$. Lines 1 to 4 can be done by a BFS traversal for each pair (n, q_0) with $n \in V_s(r)$, which is in

$$O(|V_s(r)|(|cgraph| + |cgraph|)) = O(|V_s(r)||cgraph|).$$

On the other hand, the total cost of lines 6 to 10 requires dealing with the priority queues as well as the traversal of *cgraph*, whose total cost is in

$$O(|cgraph||V_s(r)| \log(|cgraph||V_s(r)|)).$$

Therefore, if $|cgraph| < |S|$, then the total cost of Main is in

$$O(|E||\Delta| + |V_{out}(r)||E(r)||Q||\Delta| + |V_s(r)||cgraph|), \quad (4)$$

otherwise the cost is in

$$O(|E||\Delta| + |V_{out}(r)||E(r)||Q||\Delta| + |cgraph||V_s(r)| \log(|cgraph||V_s(r)|)). \quad (5)$$

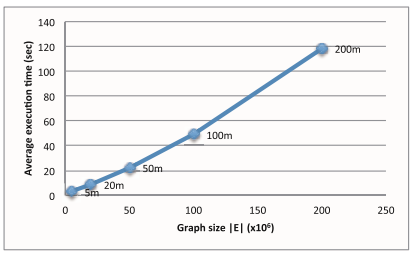
6. Experimental Results

We implemented the algorithm in C++ and made evaluation experiments on I/O cost, execution time, and the size of *cgraph*. We made two experiments under datasets generated by SP^{2}Bench [32] and DBPSB [33], respectively. These two datasets are contrasting. SP^{2}Bench generates synthetic data based on DBLP, and thus it is under a small schema and strictly structured. On the other hand, DBPSB is based on a subset of DBPedia. Thus it has no explicit schema and is more loosely structured.

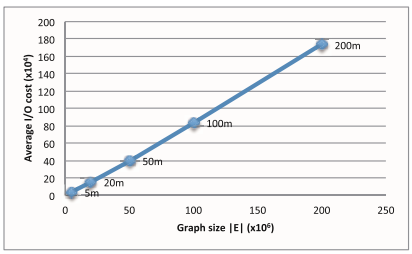
To obtain query sets for the experiments, we made a query generator for regular path queries. In short, this program generates a sequence of labels and operators randomly, and constructs regular path queries by combining these elements. However, if a sequence of labels were generated completely randomly, the answers of most queries would be empty. Therefore, the program generates a sequence of labels so that adjacent labels are "connected", e.g., a sequence "$l_1\, l_2$" is generated only if there exists at least one node n such that an incoming edge of n is labeled by l_1 and that an

Table 1 Summary of the SP^{2}Bench dataset

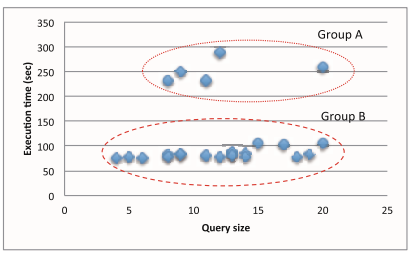
	5m	20m	50m	100m	200m
Resources (internal nodes)	911,493	3,472,940	8,698,023	17,823,525	36,447,413
Literals (leaf nodes)	2,693,483	10,506,697	25,880,271	51,298,417	101,914,699
$\|V\|$ (resources+ literals)	3,604,976	13,979,637	34,578,294	69,121,942	138,362,112
$\|E\|$	5,000,630	20,000,686	50,000,863	100,000,374	200,000,016
$\|E\|/\|V\|$	1.39	1.43	1.45	1.45	1.45
$\|\Sigma\|$	74	74	76	77	77
Size of N (GBytes)	0.14	0.57	1.46	3.04	6.28



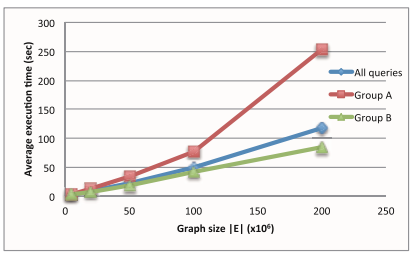
(a) Execution time



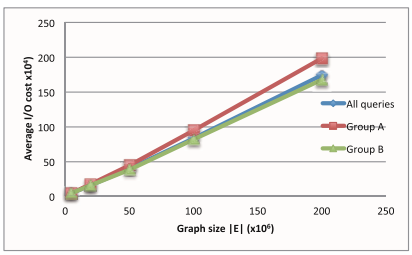
(b) I/O cost (read + write)



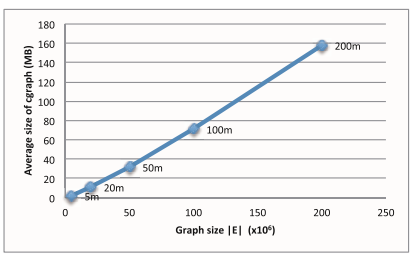
(c) Execution time of each query on the 200m graph



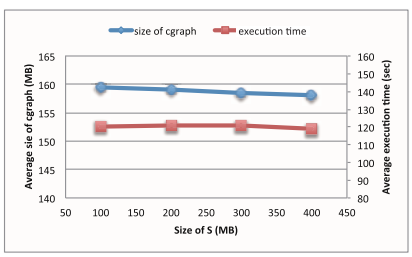
(d) Execution time for groups A and B



(e) I/O cost for groups A and B



(f) Size of *cgraph*



(g) Relation between $|cgraph|$ and $|S|$

Fig. 6 Experimental results for the SP^{2}Bench dataset

outgoing edge of n is labeled by l_2.

All the evaluations were executed on a machine with Intel Core i7 3.5GHz CPU, 8GB RAM, 2TB SATA HDD, and Linux OS (CentOS 7). In order to verify that our algorithm works in small memory, we limited the size of main memory available to each process to 4GB. Also, $|S|$ is set to 400MB unless otherwise stated. Since Linux OS has a buffer cache to store previously accessed files, we cleared the cache prior to each query execution.

6.1 Experiment for SP^{2}Bench

In the first experiment, we generated five RDF files of different sizes by SP^{2}Bench, and constructed node lists N from the RDF files. Table 1 presents a summary of the dataset. Throughout this experiment, we used 25 regular path queries generated by the above query generator. The average size of the queries is 12.0 (6.5 labels and 5.5 operators), and 16 out of the 25 queries have non-empty answers.

Figure 6 (a) plots the average execution time of the 25 queries. As shown in the figure, the execution time is almost liner to graph size $|E|$. Table 2 lists (a) the average I/O time and (b) the average execution time ((b) coincides with Fig. 6 (a)). This result suggests that the design of our algorithm, based on sequential scans of N and *cgraph*, is effective to keep the I/O time considerably small. Table 3 lists the numbers of edges that are not deleted by Prune. This shows that more than 90 percent of the edges in E are deleted by Prune.

Figure 6 (c) plots the execution times of the 25 queries

Table 2 I/O time

	5m	20m	50m	100m	200m
(a) I/O (read + write) time (sec)	0.12	0.44	1.14	2.42	5.56
(b) Execution time (sec)	2.92	8.88	22.48	49	119
(a)/(b)	0.041	0.050	0.051	0.049	0.046

Table 3 The number of edges not deleted by PRUNE

	5m	20m	50m	100m	200m
Average number of edges R not deleted by PRUNE	394369	1569674	4162446	8998095	19151186
$\|R\|/\|E\|$	0.079	0.078	0.083	0.090	0.096

Table 4 Number of queries for which $|cgraph|$ exceeds $|S|$

5m	20m	50m	100m	200m
0	0	0	0	5

on the 200m graph. 20 queries are executed in about 100 seconds (group B), while five queries require more than 200 seconds (group A). Here, we have $|cgraph| > |S|$ for the five queries in group A, and as shown in Table 4, the queries in group A are the only queries for which $|cgraph|$ exceeds $|S|$. The difference between groups A and B can be explained as follows. Some of the labels in SP^{2}Bench occur very frequently, e.g., every node has a "type" edge indicating the class in which the node belongs to, and the five queries in group A contain such "frequent" labels. As a result, if a query contains a frequent label, smaller number of edges are deleted by PRUNE and more time is required for executing the query.

Let us compare the result of this experiment and the cost estimations in Sect. 5. Figure 6 (d) plots the average execution times of the following: the 25 queries (equivalent to Fig. 6 (a)), the queries in group A, and the queries in group B. First, the execution time of group B is linear to the size of input graph, which reasonably falls below the CPU cost estimation (4). Consider next group A. Recall that the last term $|cgraph||V_s(r)|$ of the CPU cost estimation (4) changes to $|cgraph||V_s(r)|\log(|cgraph||V_s(r)|)$ when $|cgraph|$ exceeds $|S|$. This suggests that the execution time of a query noticeably increases when $|cgraph|$ exceeds $|S|$. In fact, the slope between 100m and 200m in Fig. 6 (d) becomes remarkably larger than the slope between 5m to 100m. On the other hand, as shown in Fig. 6 (e), there is no significant gap between the I/O costs of groups A and B. This suggests that the third term $|cgraph|^2$ of (2), which is added to (1) due to the fact that $|cgraph| > |S|$, may have only a small affect on the I/O cost. In summary, these results largely fit in the I/O and CPU cost estimations, but further investigations for graphs larger than the 200m graph should be made as a future work.

Consider the size of *cgraph*. Figure 6 (f) plots the average size of *cgraph* of the 25 queries. As shown in the figure, *cgraph* grows almost linearly to the size of input graph. We also examined how the size of S affects the size of *cgraph* and the execution time of the algorithm. In this examination, we used the 200m graph and changed the size of S from 100 to 400 Mbytes (Fig. 6 (g)). Both the execution time and the size of *cgraph* hardly change as the size of S changes. These results suggest that the size of S has little impact on the execution time of the algorithm unless S is extremely small.

Finally, Table 5 lists the average execution time of our algorithm and that of the algorithm in [6]† for the 25 queries. The algorithm in [6] was implemented in Java and we executed it in Java 1.7 environment. As the result, the algorithm in [6] did not work on the 20m and larger graphs due to OutOfMemory error, even if the 4GB memory limit is disabled and Java's heap size is extended to 8GB. This suggests that our external memory approach is effective to solve the all-pairs regular path problem on large graphs in a single PC environment.

6.2 Experiment for DBPSB

In the second experiment, we used four DBPSB graphs denoted "15m", "77m", "154m", "279m", named after the size of E (Table 6)††. We used the same query generator as above and generated 25 regular path queries. The average size of the queries is 11.12 (5.84 labels and 5.28 operators), and 19 queries have non-empty answers.

Figure 7 shows the results of this experiment. The overall tendency is similar to the first experiment, and thus we focus on the points different to the first one. As shown in Figs. 7 (a) to (c), the slopes of the execution time, $|cgraph|$, and the I/O cost between 154m and 279m are noticeably larger than the corresponding slopes between 15m and 154m. A possible reason for this is that the 279m graph contains more nodes, especially resources, relative to $|E|$ than the other three graphs (as shown in Table 6, the number of resources increases drastically at the 279m graph). In fact, the cost estimations (1) to (5) in Sect. 5 depend on $|V|$, $|V_{in}(r)|$, $|V_{out}(r)|$, and $|V_s(r)|$ as well as $|E|$, which largely accounts for the above increases of the execution time, $|cgraph|$, and the I/O cost at the 279m graph.

Figure 7 (d) plots the execution times of the 25 queries on the 279m graph. The 24 queries in group B are executed in about 110 seconds, while the query in group A requires 191 seconds. Here, the query in group A is the only one

†We use the implementation at https://www.informatik.hu-berlin.de/de/forschung/gebiete/wbi/resources/rpq.

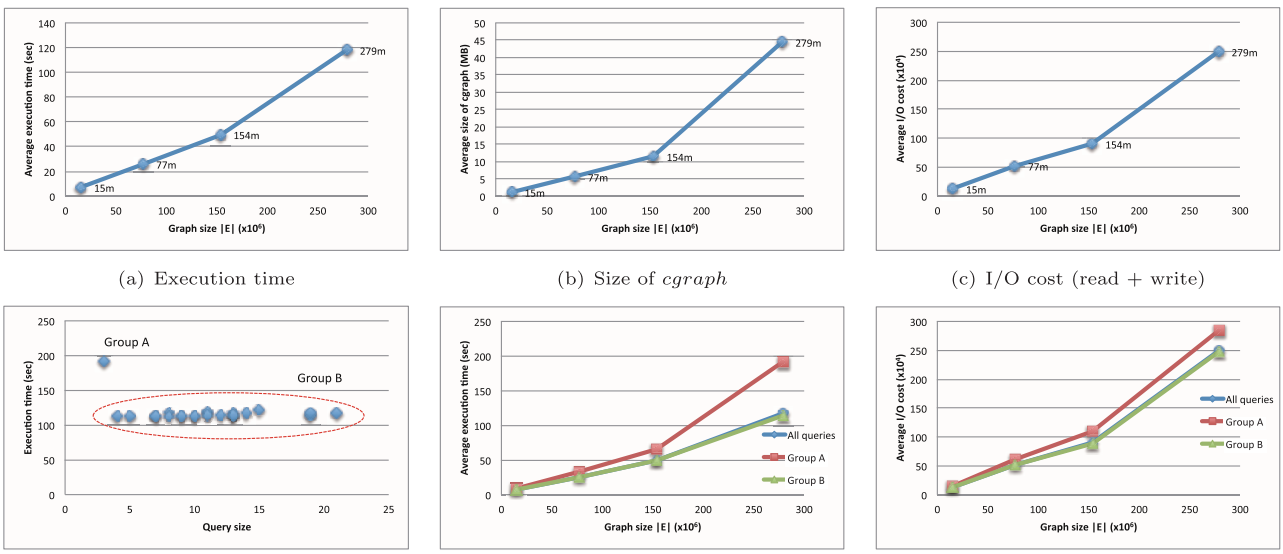
††These files are obtained from `benchmark_{10,50,100,200}.nt.bz2` available at http://benchmark.dbpedia.org/.

Table 5 Execution times of our algorithm and the algorithm of [6]

	5m	20m	50m	100m	200m
Our algorithm (sec)	2.92	8.88	22.48	49.08	118.56
The algorithm in [6] (sec)	13	-	-	-	-

Table 6 Summary of the DBPSB dataset

	15m	77m	154m	279m
Resources (internal nodes)	2,086,681	6,170,357	7,283,873	38,401,650
Literals (leaf nodes)	7,596,421	37,542,809	76,879,431	136,728,332
$\|V\|$ (resources+ literals)	9,683,102	43,713,166	84,163,304	175,130,002
$\|E\|$	15,373,842	76,868,931	153,737,801	278,913,740
$\|E\|/\|V\|$	1.59	1.76	1.82	1.59
$\|\Sigma\|$	14,130	22,147	23,344	39,675
Size of N (GBytes)	0.47	1.97	3.40	9.41



(a) Execution time (b) Size of *cgraph* (c) I/O cost (read + write)

(d) Execution time of each query on the 279m graph (e) Execution time for groups A and B (f) I/O cost for groups A and B

Fig. 7 Experimental results for the DBPSB dataset

Table 7 The number of edges not deleted by PRUNE

	15m	77m	154m	279m
Average number of edges R not deleted by PRUNE	45204	203344	407036	983968
$\|R\|/\|E\|$	0.0029	0.0026	0.0026	0.0035

for which the size of *cgraph* exceeds $|S|$, caused by a frequent label. Note that group A of this experiment consists of only one query while group A of the first experiment consists of five queries. A possible reason for this difference is that, as shown in Tables 1 and 6, the size of Σ of DBPSB is much larger than that of SP^{2}Bench, and thus a query under DBPSB is less likely to contain a frequent label.

Figures 7 (e, f) plot the execution times and the I/O costs of groups A and B. Similarly to the first experiment, the gap between the execution times of groups A and B under the 279m graph is remarkably larger, while there is little gap between the I/O costs of groups A and B.

Table 7 lists the numbers of edges not deleted by PRUNE. The ratios listed in this table are significantly smaller than those of Table 3. This is because, as shown in Tables 1 and 6, the size of Σ of DBPSB is much larger than that of SP^{2}Bench, and thus PRUNE can delete relatively more edges under the DBPSB graphs to the SP^{2}Bench graphs. Moreover, although the size of the 279m graph (DBPSB) is larger than that of 200m (SP^{2}Bench), the average execution times of the two graphs are almost the same. This implies that the execution time of a query is affected by the size of Σ as well as the size of input graph.

Finally, the algorithm of [6] did not work due to OutOfMemory error even on the 15m graph, even if the 4GB memory limit is disabled and Java's heap size is extended to 8GB.

7. Conclusion

In this paper, we proposed an external memory algorithm for the all-pairs regular path problem. Our algorithm finds the answers matching r by scanning the node list of G sequentially, which avoids random accesses to disk and thus makes regular path query processing efficient. The experiments suggest our approach is effective in solving the all-pairs regular path problem on large graphs.

However, this work has just started and we still have a lot things to do. First, we need to investigate the performance of our algorithm by using much more kinds of datasets as well as SP^{2}Bench and DBPSB. Second, we have to compare our algorithm with more graph stores. Third, we have to consider node ordering of N. For example, [34] proposes a node ordering which achieves an efficient child and descendant traversal on tree structured data. We need to consider node ordering of N that makes the algorithm more efficient.

Acknowledgments

The authors would like to thank anonymous reviewers for their valuable comments to improve this paper.

References

[1] N. Suzuki, K. Ikeda, and Y. Kwon, "An external memory algorithm for all-pairs regular path problem," Proc. DEXA 2015, vol.9262, pp.399–414, 2015.
[2] W3C, "SPARQL Query Language for RDF." http://www.w3.org/TR/rdf-sparql-query/
[3] K. Losemann and W. Martens, "The complexity of regular expressions and property paths in SPARQL," ACM Trans. Database Syst., vol.38, no.4, pp.24:1–24:39, Dec. 2013.
[4] P.B. Baeza, "Querying graph databases," Proc. PODS 2013, pp.175–188, 2013.
[5] P.T. Wood, "Query languages for graph databases," SIGMOD Rec., vol.41, no.1, pp.50–60, April 2012.
[6] A. Koschmieder and U. Leser, "Regular path queries on large graphs," Proc. SSDBM 2012, vol.7338, pp.177–194, 2012.
[7] D.C. Stefanescu, A. Thomo, and L. Thomo, "Distributed evaluation of generalized path queries," Proc. SAC 2005, pp.610–616, 2005.
[8] M. Shoaran and A. Thomo, "Distributed multi-source regular path queries," Proc. ISPA 2007, pp.365–374, 2007.
[9] L.-D. Tung, Q. Nguyen-Van, and Z. Hu, "Efficient query evaluation on distributed graphs with Hadoop environment," SoICT '13, pp.311–319, 2013.
[10] Y. Bai, C. Wang, Y. Ning, H. Wu, and H. Wang, "G-path: Flexible path pattern query on large graphs," Proc. WWW 2013, pp.333–336, 2013.
[11] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," Proc. SIGMOD 2010, pp.135–146, 2010.
[12] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, and J.L. Larriba-Pey, "Dex: High-performance exploration on large graphs for information retrieval," Proc. CIKM 2007, pp.573–582, 2007.
[13] H. Yildirim, V. Chaoji, and M.J. Zaki, "Grail: A scalable index for reachability queries in very large graphs," The VLDB Journal, vol.21, no.4, pp.509–534, Aug. 2012.
[14] L. Zou, M.T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao, "gStore: A graph-based SPARQL query engine," The VLDB Journal, vol.23, no.4, pp.565–590, Aug. 2014.
[15] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," Proc. 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pp.31–46, 2012.
[16] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC," Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '13, pp.77–85, 2013.
[17] Q. Li and B. Moon, "Indexing and querying XML data for regular path expressions," Proc. 27th International Conference on Very Large Data Bases, VLDB '01, pp.361–370, 2001.
[18] T. Milo and D. Suciu, "Index structures for path expressions," Proceedings of the 7th International Conference on Database Theory, ICDT '99, vol.1540, pp.277–295, 1999.
[19] R. Kaushik, P. Bohannon, J.F. Naughton, and H.F. Korth, "Covering indexes for branching path queries," Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD '02, pp.133–144, 2002.
[20] M. Fernandez and D. Suciu, "Optimizing regular path expressions using graph schemas," Proceedings of the Fourteenth International Conference on Data Engineering, ICDE '98, pp.14–23, 1998.
[21] F. Alkhateeb, J.-F. Baget, and J. Euzenat, "Extending SPARQL with regular expression patterns (for querying RDF)," Web Semant., vol.7, no.2, pp.57–73, 2009.
[22] L.T. Detwiler, D. Suciu, and J.F. Brinkley, "Regular paths in SparQL: Querying the NCI thesaurus," AMIA Annual Symposium, pp.161–165, 2008.
[23] K. Anyanwu, A. Maduko, and A. Sheth, "SPARQ2L: Towards support for subgraph extraction queries in RDF databases," Proceedings of the 16th International Conference on World Wide Web, WWW '07, pp.797–806, 2007.
[24] H. Zauner, B. Linse, T. Furche, and F. Bry, "A RPL through RDF: Expressive navigation in RDF graphs," Proceedings of the Fourth International Conference on Web Reasoning and Rule Systems, RR'10, vol.6333, pp.251–257, 2010.
[25] K.J. Kochut and M. Janik, "SPARQLeR: Extended SPQRQL for semantic association discovery," Proc. 4th European Conference on The Semantic Web: Research and Applications, ESWC '07, pp.145–159, 2007.
[26] X. Hu, Y. Tao, and C.-W. Chung, "Massive graph triangulation," Proc. SIGMOD 2013, pp.325–336, 2013.
[27] Z. Zhang, J.X. Yu, L. Qin, L. Chang, and X. Lin, "I/O efficient: Computing sccs in massive graphs," Proc. SIGMOD 2013, pp.181–192, 2013.
[28] Z. Zhang, J.X. Yu, L. Qin, Q. Zhu, and X. Zhou, "I/O cost minimization: Reachability queries processing over massive graphs," Proc. EDBT 2012, pp.468–479, 2012.
[29] Y. Luo, G.H.L. Fletcher, J. Hidders, Y. Wu, and P. De Bra, "External memory k-bisimulation reduction of big graphs," Proc. CIKM 2013, pp.919–928, 2013.
[30] P. Buneman, M. Fernandez, and D. Suciu, "UnQL: A query language and algebra for semistructured data based on structural recursion," The VLDB Journal, vol.9, no.1, pp.76–110, 2000.
[31] A. Aggarwal and J. Vitter, "The input/output complexity of sorting and related problems," Commun. ACM, vol.31, no.9, pp.1116–1127, Sept. 1988.
[32] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP2Bench: A SPARQL performance benchmark," Proc. ICDE 2009, pp.222–233, 2009.
[33] M. Morsey, J. Lehmann, S. Auer, and A.-C.N. Ngomo, "DBpedia SPARQL benchmark – Performance assessment with real queries on real data," Proceedings of the 10th International Conference on The Semantic Web - Volume Part I, ISWC'11, vol.7031, pp.454–469,

2011.
[34] A. Morishima, K. Tajima, and M. Tadaishi, "Optimal tree node ordering for child/descendant navigations," Proc. ICDE 2010, pp.840–843, 2010.

Appendix: I/O and CPU Costs of the Algorithm

Let $G = (V, E, \Sigma)$ be a graph, r be a regular path query, and $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA such that $L(A) = L(r)$.

A.1 I/O Cost

We assume that data is transferred between HDD and main memory in blocks of size B.

Consider first the cost of constructing node list N. To construct node list N, for each node $n \in V$, we have to collect $Out(n)$, and $inMax(n, l)$ and $inMin(n, l)$ for each label l. Collecting $Out(n)$ of all the nodes n can be done in $O(scan(|E|))$. Collecting $inMax(n, l)$ and $inMin(n, l)$ of all the nodes n can be done by sorting E w.r.t. the target node of each edge and then scanning the sorted edge list, which requires $O(sort(|E|)+scan(|E|)) = O(sort(|E|))$ I/O cost. The nodes with collected information are inserted into an external B+-tree and then exported as node list N, which can be done in $O(|V| \log_B |V|)$. Thus the I/O cost of constructing N is

$$O(sort(|E|) + |V| \log_B |V|).$$

Let us next consider the I/O cost of the algorithm. Since node list N contains $Out(n)$, $order(n)$, and $inMax(n, l)$ and $inMin(n, l)$ for each node n, the size of N is in $O(|E| + |V|)$. Thus, in Main, the I/O cost of reading N sequentially is in $O((|E| + |V|)/B)$. Moreover, the cost of writing *cgraph* is $O((|cgraph|)/B)$. Suppose that the size of *cgraph* does not exceed $|S|$. Then TraverseCgraph reads *cgraph* only once, where $|cgraph| < |S| < |N|$. Let Ans be the set of answers. Taking the cost of outputting Ans into account, the total I/O cost of Main is

$$O((|E| + |V| + output(|Ans|))/B).$$

Here, $|Ans|$ is in $O(|V_s(r)||V_a(r)|)$, where V_s is the set of "possible start nodes", that is,

$$V_s(r) = \{n \in V \mid n \xrightarrow{l} n' \in Out(n), \delta(q_0, l) \neq \emptyset\},$$

and $V_a(r)$ is the set of "possible accepting nodes", that is,

$$V_a(r) = \{n' \in V \mid n \xrightarrow{l} n' \in E, \delta(q, l) \cap F \neq \emptyset \text{ for some } q \in Q\}.$$

Then consider the case where the size of *cgraph* exceeds $|S|$. Each ScanForward/ScanBackward execution requires $O(|cgraph|/B)$ I/O read cost. Lines 7 to 10 of TraverseCgraph are repeated at most $|cgraph|$ times (the length of the possible longest answer path is $|cgraph|$, which is a very extreme case although). Therefore, the I/O read cost of TraverseCgraph is in $O(|cgraph|^2/B)$. Hence the total I/O cost of Main in this case is

$$O((|E| + |V| + |cgraph|^2 + output(|Ans|))/B).$$

Let us consider the size of *cgraph*. Let $E(r) = \{n \xrightarrow{l} n' \in E \mid l \text{ appears in } r\}$ and $\Delta = \{(q, q') \in Q \times Q \mid q' \in \hat{\delta}(q, w), w \in \Sigma^*\}$. Let us consider how many edges are created and added to *cgraph* by Main. Main "partitions" N into $k = \lceil \frac{|N|}{|S|} \rceil$ sublists. In the following, we assume that the number of boundary edges in N is proportional to k. Let N_i be the i-th sublist of the k sublists, and consider the following sets of edges generated by Main when N_i is loaded into S. We write $n \in N_i$ if node n is contained in N_i.

- $E_1(N_i)$ is the set of edges between T_{in} and T_{out}. That is, $(n, q) \to (n'', q'') \in E_1(N_i)$ iff $T_{in}(n, q)$, $T_{out}(n', q')$, N_i contains a path $n \rightsquigarrow_p n'$, $q' \in \hat{\delta}(q, l(p))$, $n' \xrightarrow{l} n'' \in Out(n')$, $q'' \in \delta(q', l)$, and $n'' \notin N_i$.
- $E_2(N_i)$ is the set of edges between T_{start} and T_{out}. That is, $(n, q) \to (n'', q'') \in E_2(N_i)$ iff $T_{start}(n, q)$, $T_{out}(n', q')$, N_i contains a path $n \rightsquigarrow_p n'$, $q' \in \hat{\delta}(q, l(p))$, $n' \xrightarrow{l} n'' \in Out(n')$, $q'' \in \delta(q', l)$, and $n'' \notin N_i$.
- $E_3(N_i)$ is the set of edges between T_{in} and T_{accept}. That is, $(n, q) \to (n', q') \in E_3(N_i)$ iff $T_{in}(n, q)$, $T_{accept}(n', q')$, N_i contains a path $n \rightsquigarrow_p n'$, and $q' \in \hat{\delta}(q, l(p))$.

Consider $E_1(N_i)$. Let

$$V_{in}(r) = \{n \in V \mid n' \xrightarrow{l} n \in E(r)\}.$$

Moreover, let

$$V_{in}(N_i) = \{n \in N_i \mid n' \xrightarrow{l} n \in E(r), n' \notin N_i\}.$$

Then $|V_{in}(N_i)|$ is in $O(|V_{in}(r)|/k)$. Let

$$E_{out}(N_i) = \{n' \xrightarrow{l} n'' \in E(r) \mid n' \in N_i, n'' \notin N_i\}.$$

By the above assumption, the number of boundary edges in $E(r)$ is $\epsilon k|E(r)|$ for some $\epsilon < 1$. Thus $|E_{out}(N_i)|$ is in $O((\epsilon k|E(r)|)/k) = O(|E(r)|)$. Let

$$D(N_i) = \{(q, q') \mid T_{in}(n, q), T_{out}(n', q'), q' \in \hat{\delta}(q, w), w \in \Sigma^*, n, n' \in N_i\}.$$

Then $|D(N_i)|$ is in $O(|\Delta|)$. Thus $|E_1(N_i)|$ is in

$$O(|V_{in}(N_i)||E_{out}(N_i)||D(N_i)|) = O\left(\frac{|V_{in}(r)||E(r)||\Delta|}{k}\right). \quad (A\cdot 1)$$

We can also show that $E_2(N_i)$ and $E_3(N_i)$ are in (A· 1). Thus $|cgraph|$ is in

$$O(\sum_k (|E_1(N_i) + E_2(N_i) + E_3(N_i)|)) = O(|V_{in}(r)||E(r)||\Delta|). \quad (A\cdot 2)$$

Thus, *cgraph* is not too large whenever a number of unnecessary edges are deleted by Prune.

In fact, this is still an overestimation and in general the size of *cgraph* is much smaller than (A·2). For example, consider $E_1(N_i)$. Let $n \in V_{in}(N_i)$, $n' \xrightarrow{l} n'' \in E_{out}(N_i)$, and $(q, q') \in D(N_i)$ such that $T_{in}(n, q)$ and that $T_{out}(n', q')$. Then the above estimation assumes that N_i "always" contains a path $n \rightsquigarrow_p n'$ such that $q' \in \hat{\delta}(q, l(p))$ whenever $T_{in}(n, q)$ and $T_{out}(n', q')$. However, in general the probability of existing such a path from n to n' is very low and thus $|E_1(N_i)|$ is significantly smaller than (A·1). Identifying a tighter upper bound is left as a future work.

A.2 CPU Cost

Let us next consider the CPU cost of the algorithm. Since N is partitioned into $k = \left\lceil \frac{|N|}{|S|} \right\rceil$ sublists, lines 5 to 9 of Main are repeated k times. For each sublist, since the number of edges in S is in $O(|E|/k)$, Prune in line 7 can be done in $O(|E||\Delta|/k)$ time. Consider TraverseBuf called in line 8 of Main. Let

$$V_{out}(r) = \{n \in V \mid n \xrightarrow{l} n' \in E(r)\}.$$

Then the for loop in line 2 of TraverseBuf repeats $O(|V_{out}(r)||Q|/k)$ times, and lines 3 to 12 can be done by a BFS traversal over the edges in S plus the edges leaving S, which is in $O(|E(r)||\Delta|)$. Thus, for each sublist of N loaded into S, the cost of TraverseBuf is in $O((|V_{out}(r)||Q|/k)|E(r)||\Delta|) = O(|V_{out}(r)||E(r)||Q||\Delta|/k)$. Thus, the cost of Main except TraverseCgraph is in

$$\begin{aligned} &O(k(|E||\Delta|/k + |V_{out}(r)||E(r)||Q||\Delta|/k)) \\ &= O(|E||\Delta| + |V_{out}(r)||E(r)||Q||\Delta|). \end{aligned} \quad (A\cdot 3)$$

Consider the cost of TraverseCgraph. Then the numbers of nodes and edges in *cgraph* are in $O(|cgraph|)$. Lines 1 to 4 can be done by a BFS traversal for each pair (n, q_0) with $n \in V_s(r)$, which is in

$$\begin{aligned} &O(|V_s(r)|(|cgraph| + |cgraph|)) \\ &= O(|V_s(r)||cgraph|). \end{aligned} \quad (A\cdot 4)$$

On the other hand, the total cost of lines 6 to 10 is the sum of

1. the cost of traversing nodes in *cgraph* for each pair (n, q_0) with $n \in V_s(r)$, and
2. the cost of adding/deleting pairs to/from the priority queues.

The cost of 1 is in $O(|V_s(r)||cgraph|)$ and the cost of 2 is in $O(|cgraph||V_s(r)| \log(|cgraph||V_s(r)|))$. Thus the total cost of lines 6 to 10 is in

$$O(|cgraph||V_s(r)| \log(|cgraph||V_s(r)|)). \quad (A\cdot 5)$$

Consequently, if $|cgraph| < |S|$, then by (A·3) and (A·4) the total cost of Main is in

$$O(|E||\Delta| + |V_{out}(r)||E(r)||Q||\Delta| + |V_s(r)||cgraph|).$$

Otherwise, by (A·3) and (A·5) the cost is in

$$\begin{aligned} &O(|E||\Delta| + |V_{out}(r)||E(r)||Q||\Delta| \\ &+ |cgraph||V_s(r)| \log(|cgraph||V_s(r)|)). \end{aligned}$$

Nobutaka Suzuki received his B.E. degree in information and computer sciences from Osaka University in 1993, and his M.E. and Ph.D. degrees in information science from Nara Institute of Science and Technology in 1995 and 1998, respectively. He was with Okayama Prefectural University as a Research Associate in 1998–2004. In 2004, he joined University of Tsukuba as an Assistant Professor. Since 2009, he has been an Associate Professor of Graduate School of Library, Information and Media Studies, University of Tsukuba. His current research interests include database theory and structured documents.

Kosetsu Ikeda received his bachelor's degree in library and information science from University of Tsukuba in 2011, and his M.E. degree in information science from Tsukuba University in 2013. He has been an Ph.D. student of Graduate School of Library, Information and Media Studies, University of Tsukuba. His current research interests are XML query processing and Web data management.

Yeondae Kwon is an Assistant Professor at University of Tokyo. She received her M.S. degree in chemistry from Pusan National University, Busan, Korea, and Ph.D. degree in information science from Nara Institute of Science and Technology, Nara, Japan, in 2000. Her research interests include text mining of biomedical literature and healthcare data mining. She is a member of the IPSJ.