International Journal of Software Engineering and Its Applications
Vol.8, No.2 (2014), pp.343-358
http://dx.doi.org/10.14257/ijseia.2014.8.2.34

A RDBMS based Framework for Shortest Path Discovery with Constraint Paths

Jihye Hong[1], Yongkoo Han[2] and Young-Koo Lee[3,*]

[1,2,3]*Kyung Hee University*
[1]*hjhh@khu.ac.kr*, [2]*ykhan@khu.ac.kr*, [3]*yklee@khu.ac.kr*

Abstract

Nowadays, graphs are used in many applications and the graph size is rapidly increasing. The shortest path discovery considering constraint conditions is very important. For example, when a road is temporarily not available, paths not including the invalid road are required. In this paper, we propose an efficient RDBMS based shortest path discovery method considering constraint conditions in a large graph. The proposed method finds the shortest path not containing the constraint path by avoiding expanding the paths including the given constraint path. We propose a novel approach that expands strategy by using the index table proposed by the FEM framework. We also propose an optimizing method for multiple constraint paths. We experimentally verify that our approaches can achieve high space and time efficiency in a large graph.

Keywords: *Shortest Path Searching, RDBMS, Graph Searching, Constraint Condition*

1. Introduction

Graphs provide a way to represent relationships between objects, where some pairs of objects are connected by links. Graphs are used in many applications such as social networks, web page links, telephone networks, and ontology graphs. Accordingly, many graph algorithms such as the shortest path discovery [1-4], minimal spanning tree [5], and the path for traveling salesman [6] are developed to analyze these networks effectively and efficiently.

The shortest path discovery considering constraint conditions is an important graph search query. The shortest path discovery plays a key role in many networks and the networks often contain constraint conditions in the real world. For example, when a road is temporarily not available, paths not including the invalid road are required. Another example is the telephone network containing link failure. Similarly, this kind of constraint can occur in many networks.

Nowadays, network sizes are rapidly increasing, and the networks cannot fit into the memory. Therefore, memory based algorithms [7-8] for shortest path discovery considering constraints have degradation in performance or cannot work well in the large-scale networks. There are various disk based approaches [9-10] to support large-scale graphs. A single computer based approach [9] divides a large graph into memory sized subgraphs and loads them into the memory to process partially. The other main stream is a MapReduce based distributed processing approach [11-12]. This approach stores large graphs in the distributed file system over a cluster of computers and processes them in parallel. However, accessing graphs is difficult because MapReduce framework [11-12] does not fully support schema and index mechanism.

* Corresponding author

ISSN: 1738-9984 IJSEIA

Recently, a RDB based graph processing approach [13-16] has been proposed. The relational database (RDB) is an effective data management approach for large scale and complex data because RDB provides a stable infrastructure and several graph related functions such as breadth-first-search (BFS) and graph reachability operations. HDB-SUBDUE [13] and DB-FSG [14] proposed a RDB based frequent subgraph mining method. Gao *et al.*, [15] proposed a generic Frontier-Expansion-Merge (FEM) framework for graph search operations in RDB context, and implemented the shortest path discovery on the framework. In order to improve the performance, the FEM framework uses an index table that stores pre-computed partial paths.

However, there is no existing work shortest path discovery considering constraint conditions in RDB. It is also difficult to consider constraint conditions on the FEM framework. The index table maintains only partial paths, which enables us to trace the shortest path efficiently. We cannot find the entire paths including the given constraint paths directly from the index table.

In this paper, we propose an efficient RDB based method that finds a shortest path with constraint conditions in a large graph. The proposed method searches the shortest path not containing the constraint paths by avoiding expanding the paths including the given constraint path. We propose an efficient expanding strategy that uses the index table proposed by the FEM framework. In order to optimize the performance for multiple constraint paths, we perform the proposed method one time for the constraint paths that shares common expanding paths. From the experimental results, we show that the proposed method can find a shortest path with high space efficiency in a large graph.

The remainder of the paper is organized as follows. We briefly introduce the existing graph search studies as related works in Section 2. We present terminologies used in this paper and FEM framework as a background of our work in Section 3. In Section 4, we explain the proposed shortest path searching algorithm considering a constraint condition in detail. Section 5 presents the experimental results of the proposed algorithm, and Section 6 finally concludes this paper.

2. Related Work

Shortest path discovery is very fundamental and important in graph applications. Conventional algorithms assumed that an entire graph resided in the memory. Dijkstra's algorithm [1] is the most popular algorithm in the shortest path discovery. Bi-directional search strategy [2] reduced the search space by running forward and backward searches simultaneously. The indexing techniques [3-4] improved the performance in running time by pre-computing the shortest path. However, these methods are available when the graph is fit into the memory.

As the graph size is getting larger and larger, disk-based approaches have been studied. Yuan *et al.*, [9] partitioned a large graph into the memory sized segments and discovered the shortest path, sequentially. Hutchinson *et al.*, [10] designed the index for shortest path discovery on external memory in a planar graph. Recently, MapReduce based distributed frameworks [11-12] have been studied to analysis big data. However, the MapReduce paradigm causes expensive IO cost for graph algorithms because the graph algorithms require much iteration.

Recently, RDB based graph algorithms [13-15] have been proposed. HDB-SUBDUE [13] and DB-FSG [14] proposed a RDB based frequent subgraph mining method. Gao *et al.*, [15] proposed a generic Frontier-Expansion-Merge (FEM) framework for graph search operations using three corresponding operators in the relational database (RDB) context. Wang *et al.*, [16]

proposed that RDB based frequent pattern mining from a well-known frequent pattern mining called *gSpan* [17].

Memory based shortest path searching methods considering constraint conditions have been proposed. Ahmed *et al.*, [7] proposed an incremental method that runs Dijkstra's algorithm incrementally after replicating vertices when a constraint is discovered. Villeneuve *et al.*, [8] proposed a method that avoids the constraint path by pre-computing k-shortest paths. These memory based approaches cannot be applied for large-scale graphs. Moreover, to the best of our knowledge, there is no existing searching shortest path approach considering constraint conditions in RDB.

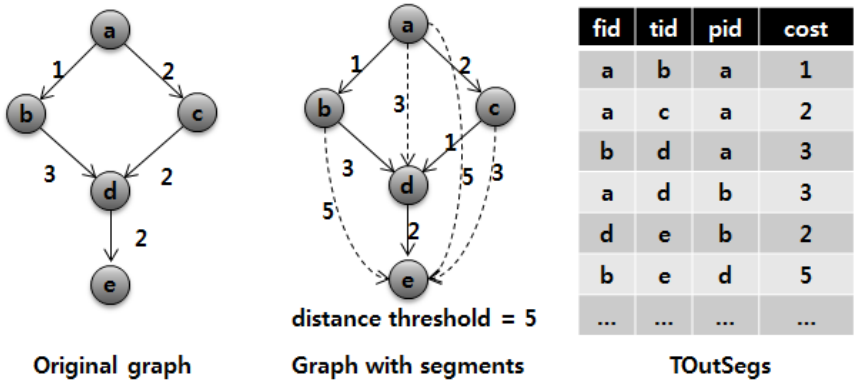
3. Graph Searching Framework based on RDB

Graph searching is widely used for graph algorithms finding specific subgraphs such as the shortest path and the graph reachability. Most of graph searching algorithms share a generic search process that iteratively extends nodes having results of query with high possibilities. For efficient implementing the generic search process, the FEM framework proposed three basic SQL operators: Frontier operator, Expand operator, and Merge operator. F-operator selects next expanding nodes, called frontier nodes, from the visited nodes. E-operator expands the frontier nodes. M-operator merges the new expanded nodes into the visited nodes.

Shortest path searching generally adopts breadth first search (BFS) to traverse a graph. BFS can only reduce the search space in the case that shortest path has a small number of nodes. A large-scale graph must have a long shortest path. Therefore, BFS requires a large number of iterative expansions in a large scale graph.

The FEM framework also requires BFS to expand all of edges of frontier nodes. For the efficient searching, the FEM framework pre-computes *shortest segments* p^{ss} with their distances shorter than the given *distance threshold* l_{thd} and stores the shortest segments into an index table called *SegTable*. Two kinds of index tables are maintained such as ToutSegs and TInSegs since the FEM framework performs bi-directional expansion. The index tables consist of source nodes (fid), target nodes (tid), parent nodes of target nodes (pid), distances of the shortest segments (cost). By expanding shortest segments in the index table, we can reduce unnecessary re-expanding for the path segments contained in the shortest segments. Therefore, we can meet the termination condition of the searching quickly.

Figure 1 shows an example of the index table. The ToutSegs table is the index table containing shortest segments of out-edges from the original graph. The graph with shortest segments is a graph representing all shortest segments as dotted-paths. If the distance threshold l_{thd} is set to 5, all shortest segments having distances shorter than 5 are stored into the index table. The dotted-path $a \rightarrow e$ with cost 5 is a pre-computed shortest segment. If we start path searching from a, e can be found in one expansion instead of three expansions such as $a \rightarrow c \rightarrow d \rightarrow e$.



fid	tid	pid	cost
a	b	a	1
a	c	a	2
b	d	a	3
a	d	b	3
d	e	b	2
b	e	d	5
...	...	...	...

Figure 1. Example of an Index Table

4. A RDB based Framework for Shortest Path Discovery with Constraint Conditions

In this section, we explain the RDB based framework that searches the shortest path considering constraint path. We first define notations used in our method in Section 4.1. We propose an efficient search method avoiding expanding constraint paths in Section 4.2, and optimize the proposed method for the multiple constraint paths in Section 4.3.

4.1. Notations

We define the novel notations related to constraint conditions.

Definition 1. Constraint path Given the shortest path, $p^s = s \rightarrow n_1 \rightarrow \cdots \rightarrow n_k \rightarrow t$, We call a path p an *constraint path* if p cannot be included in p^s. We denote the constraint path as p^c.

Definition 2. Constraint segment Given the shortest segment, $p^{ss} = s \rightarrow n_1 \rightarrow \cdots \rightarrow n_k \rightarrow t$, We call a shortest segment *constraint segment* if the shortest segment includes any constraint path. We denote the constraint path as p^{cs}

Definition 3. Subpath Given the two paths, p_A and p_B, we call p_A a *subpath* of p_B if all paths of p_A are in p_B. We denote the subpath relationship between p_A and p_B as $p_A \subseteq p_B$.

Definition 4. Proper subpath Given the two paths, $p_A = s \rightarrow n_1 \rightarrow \cdots \rightarrow n_k \rightarrow t$ and $p_B = v \rightarrow \cdots \rightarrow s \rightarrow \cdots \rightarrow t \rightarrow \cdots \rightarrow w$, we call p_A a *proper subpath* of p_B If $p_A \subseteq p_B$, and $s \neq v$ and $t \neq w$. We denote the proper subpath relationship between p_A and p_B as $p_A \subset p_B$.

Definition 5. Alternative path Given the two paths, $p_A = s \rightarrow \cdots \rightarrow t$ and $p_B = q \rightarrow \cdots \rightarrow r$, we call p_A (or p_B) an *alternative path* of p_B (or p_A) if $s = q$ and $t = r$, and $cost(p_A) = cost(p_B)$. $cost(p)$ is a function summing up the weights of all constituent edges in the path p. We denote the alternative path relationship between p_A and p_B as $p_A \approx p_B$.

Figure 1 shows the examples of the notations related to constraint conditions. The graph G contains the constraint path $p^c = b \rightarrow c$. p_1, p_2, and p_3 are constraint segments because they contains p^c. p^c is a subpah of p_1, p_2, and p_3. But only p^c is a proper subpath of p_3 because p_1 and p_2 have p^c at either of the starting or end edge. p_3 and p_4

are alternative paths to each other because they have the same starting nodes, destination nodes, and distances.

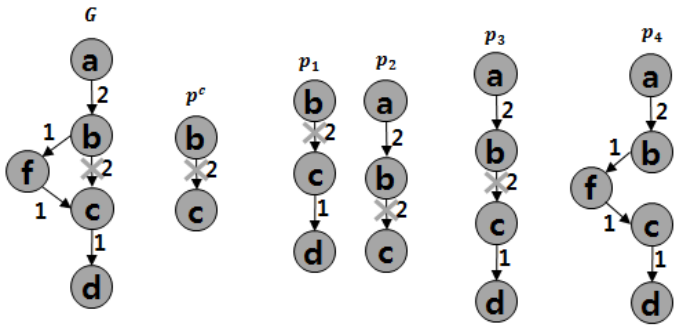


Figure 2. Examples of Notations Related to Constraints

4.2. Shortest Path Discovery Containing Constraints

In order to find all constraint segments, all edges of shortest segments are required. In the FEM framework, the index table only stores the edges connected to destination nodes. Therefore, we cannot directly avoid constraint segments from the index table if the constraint path is a proper subpath of a constraint segment. We can keep all edges of each shortest segment in the index table for solve this problem. However, this naïve approach incurs inefficiency since it requires additional $m \times n$ times space, where m is the average number of path segments per each edge and n is the number of edges. The larger index space requires more I/O cost.

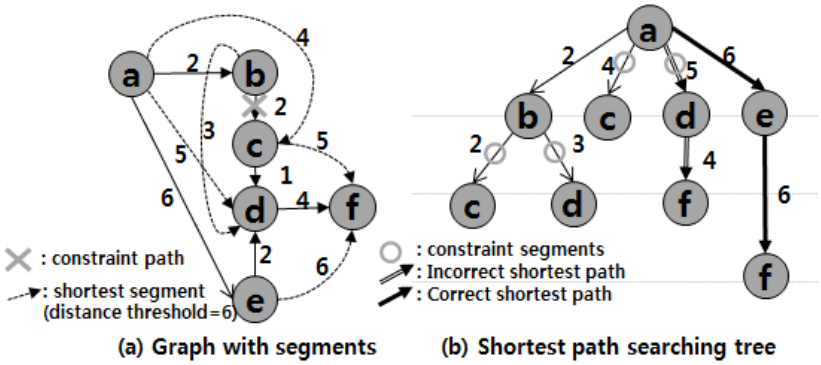


Figure 1. Problem of Shortest Path Discovery with Constraints from an Index Table

Figure 3 is an example showing the problem of shortest path discovery with constraints from the index table. Assume a shortest path discovery from the node a to the node f when the constraint path is $p^c = b \rightarrow c$ in Figure 3(a). If the constraint path p^c is not a proper subpath of a shortest segment to be expanded, we can find this shortest segment directly in the index table. We easily know that the shortest segment $p^{ss} = a \rightarrow b \rightarrow c$ includes p^c because this p^{ss} is stored as $(a, c, b, 4)$ in the index table. However, in the case that the constraint path p^c is a proper subpath of a shortest segment to be expanded such as $p^{ss} = a \rightarrow b \rightarrow c \rightarrow d$, the expended shortest segment can include constraint paths because

we cannot check the constraint segments in the middle of the shortest segments. In Figure 3(b), p^c is a proper subpath of shortest segment $a \rightarrow d$ because the full path of $a \rightarrow d$ is $a \rightarrow b \rightarrow c \rightarrow d$. However, it is difficult to figure out $a \rightarrow b \rightarrow c \rightarrow d$ from $a \rightarrow d$ in the index table. Therefore, we can select the wrong shortest path $a \rightarrow d \rightarrow f$ rather than correct one $a \rightarrow e \rightarrow f$.

We discover the shortest path with constraint paths by avoid expanding shortest segments including constraint paths. We first find all constraint segments in the index table. Then, we update the cost of these constraint segments with infinite value to avoid constraint paths. In this manner, we can compute the constraint query efficiently without modifying the shortest path algorithm and the FEM framework.

We explain how to find all constraint segments from the index table efficiently. We find all the constraint segments only in 2 iterations. In the first iteration, we find all the possible shortest segments starting from both ends of the constraint paths in backward and forward directions. In backward direction, we fist scan the index table to find shortest segments $p^{ss} = \mathrm{s} \rightarrow \cdots \rightarrow t$ connected to constraint paths $p^c = t \rightarrow \cdots \rightarrow r$. The shortest segments $s \rightarrow \cdots \rightarrow r$ must be constraint segments if the distance from s to r is short than the distance threshold l_{thd}. These processes are performed similarly in forward direction. In the second iteration, all constraint segments are generated by combining the constraint segments discovered by the forward and the backward directions.

We temporally maintain three tables: FwdSegments, BckSegments, and ConstSegments. These tables store the constraint segments in the forward, the backward, and both directions, respectively. Since each query requests different constraint, these tables are removed after processing the query. Moreover, these tables occupy much smaller space compared with the index table.

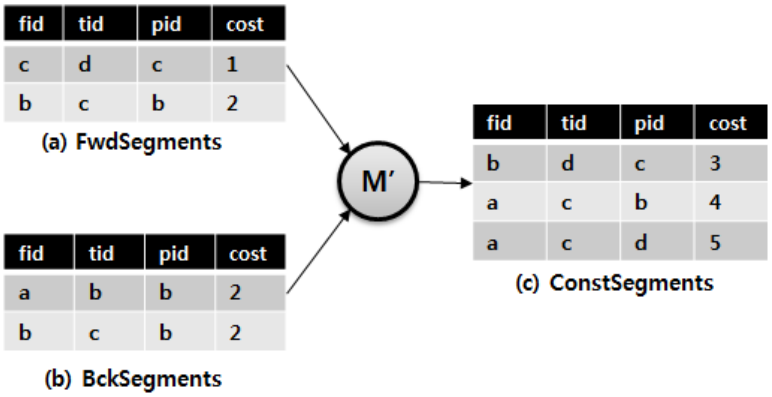


Figure 4. Searching Segment Considering a Constraint Path

Figure 4 shows an example of searching constraint segments when the constraint path is $p^c = b \rightarrow c$ in Figure 3(a). We construct a FwdSegments table with the by finding all shortest segment starting from node c. The shortest segments must have a shorter distance than the difference between the distance threshold and distance of the constraint path. For example, the shortest segment $p_1^{ss} = c \rightarrow d$ is stored in the FwdSegments table because the distance of p_1^{ss} is less than $l_{thd} - cost(p^c)$, where p_1^{ss} is 1, l_{thd} is 6, and $cost(p^c)$ is 2. In this way, we also construct the BckSegments table. After constructing two tables, we construct the ConstSgments by combining the

segments from the FwdSegments and the BckSegments. We can find shortest segments, where constraint paths are the proper subpaths of shortest segments. For example, the shortest segment $p_2^{ss} = a \to \cdots \to d \to c$ cannot be found in index table directly since p^c is the proper subpaths of p_2^{ss}.

We give the complete algorithm of searching all segments including a constraint path $p^c = u \to \cdots \to v$. Above all, we check that whether the constraint path is longer than the distance threshold or not. If so, return an empty table (line 1). First, we find all the segments having distance shorter than l_{thd} from u where l_{thd} is the index threshold and p^c is the constraint path using the SQL statement in Listing 1(1) (line 3). Second, we store all the segments found in first step on FwdSegments table (line 4). Similar to the second step, we can store the segments from u in BckSegments table (line 5). Third, we combine one segment in FwdSegments and another segment in BckSegments. In this way, we can consider all combinations having distance of combined segment shorter than l_{thd} using the SQL statement in Listing 1(4) (line 6-8). If a distance is satisfied with the given threshold, we insert the combined segment into the output set (line 9). Finally, we update the cost of constraint segments value as infinite for avoiding expanding these segments (line 10).

Algorithm 1. Searching constraint segments

Algorithm 1 SearchSegments(constraint path p^c)

•**Input:** constraint path $p^c = a \to b$
•**Output:** S = {s|∀s ∈ Index Table, $p^c \in s$}

1: **IF** distance of p^c is longer than l_{thd}
2: **RETURN** ∅;
3: **EXPAND** node v using threshold $l_{thd} - cost(p^c)$
4: **CREATE** a temporary table FwdSegments and Insert all rows through line1 to FwdSegments, S
5: Similar actions from line 1 to line 2 for the backward expansion;
6. **FOR** each segment s_1 **IN** FwdSegments
7. **FOR** each segment s_2 **IN** BckSegments
8. **IF** cost(s_1) + cost(s_2) + $cost(p^c) \leq l_{thd}$
9. **Insert** $s_1 + s_2$ into S
10. **UPDATE** the cost of constraint segments value as infinite
11. **RETURN** S

Listing 1. SQL in Searching Constraint Segments

1: Construction a FwdSegments in forward directions for a constraint path $\mathbf{p^c} = u \to \cdots \to v$
CREATE VIEW FwdSegments (fid, tid, pid, cost) **AS**
SELECT q.srcid, q.nid, q.p2s, q.dist
FROM TOutSegs q
WHERE q.srcid= v and q.dist<=$l_{thd} - cost(p^c)$
2: Construction a constraint segments table (1) : Forward segment + $\mathbf{p^c}$
CREATE TABLE ConstraintSegments(fid, tid, pid, cost) as
SELECT u, q.tid, q.fid, q.cost+ $cost(p^c)$
FROM FwdSegments q
3: Construction a constraint segments table (2) : Backward segment+$\mathbf{p^c}$
INSERT INTO ConstraintSegments (fid, tid, pid, cost)
SELECT q.fid, v, u, q.cost+ $cost(p^c)$
FROM BckSegments q
4: Construction a constraint segments table (3) : Forward segment+$\mathbf{p^c}$ + Backward segment
INSERT INTO ConstraintSegments (fid, tid, pid, cost)

```
SELECT q.fid, p.tid, p.pid, q.cost+p.cost
FROM BckSegments q, FwdSegments p
WHERE q.cost+p.cost+ cost(p^c)<=l_thd
5: Update the cost of constraint segments as infinite
MERGE INTO TOutSEgs s USING
(SELECT fid, tid
FROM ConstraintSegments ) c
ON (c.fid = s.fid and c.tid = s.tid )
WHEN MATCHED THEN UPDATE
SET cost=∞
```

4.3. Optimization for Multiple Constraints

In the real world network such as road networks, the shortest path queries often contain multiple constraint paths. For example, many roads are not available at the same time due to various road problems such as rush-hour traffic and road maintenance. In an optical network, similar situations are occurred. When a ray of light should reach a particular point, it often fails to reach the endpoint because of several transmission impairments such as crosstalk, non-linearity, and attenuation [8, 18-19]. For considering multiple constraint paths, we can perform Algorithm 1 iteratively. However, this approach requires a long running time because duplicated searches for the same constraint segments are performed.

We propose an efficient method for multiple constraints. If two paths are alternative paths to each other, all shortest segments including each path must be alternative paths to each other. This theorem is formally defined and proofed in Theorem 1. Therefore, we find the shortest segments for only one of alternative paths. The proposed approach can process the query including multiple constraints by performing algorithm 1 only once when the multiple constraints are alternative paths.

Theorem 1. Given the two paths, p_1 and p_2, the shortest segment of p_1 must have the shortest segment of p_2 as alternative segment If p_1 and p_2 are alternative paths to each other.

Proof. We prove it by contradiction. Suppose that there exists a shortest segment of p_1 that does not have any shortest segment of p_2 as an alternative path. We denote the alternative paths p_1 and p_2 as $p_1 = m \rightarrow p_{sub} \rightarrow n$ and $p_2 = m \rightarrow p'_{sub} \rightarrow n$, respectively. We can represent the shortest segment including p_1 as $p^{ss} = \text{a} \rightarrow \cdots \rightarrow p_1 \rightarrow \cdots \rightarrow b$. A path $p = a \rightarrow \cdots \rightarrow p_2 \rightarrow \cdots \rightarrow b$ must exist since p_1 and p_2 are alternative paths to each other. By the way, we have $cost(p_1) \leq cost(p^{ss}) \leq l_{thd}$ since cost(p^{ss}) is shorter than the distance threshold l_{thd}. The equation cost(p^{ss}) $= cost(p) \leq l_{thd}$ is also valid because $cost(p_1)$ is same as $cost(p_2)$ by the definition of the alternative path. Accordingly, p is the shortest segment of p_2 since $cost(p) \leq l_{thd}$. This contradicts the assumption. Therefore, the shortest segment of p_1 always has the shortest segment of p_2 as alternative segment If p_1 and p_2 have the alternative path relation.

In the naïve approach that performs Algorithm1 iteratively, the number of edge scans is $ave_{seg} \times ave_{node} \times n_{const}$, where ave_{seg} is the average number of shortest segments per each edge, ave_{node} is the average number of nodes per each shortest segment, and n_{const} is the number of given constraint paths. In the proposed approach, the number of edge scans is $ave_{seg} \times ave_{node} \times n_{alter}$, where n_{alter} is the number of distinct alternative paths. The performance of the proposed approach is guaranteed since we have $n_{alter} \leq n_{const}$.

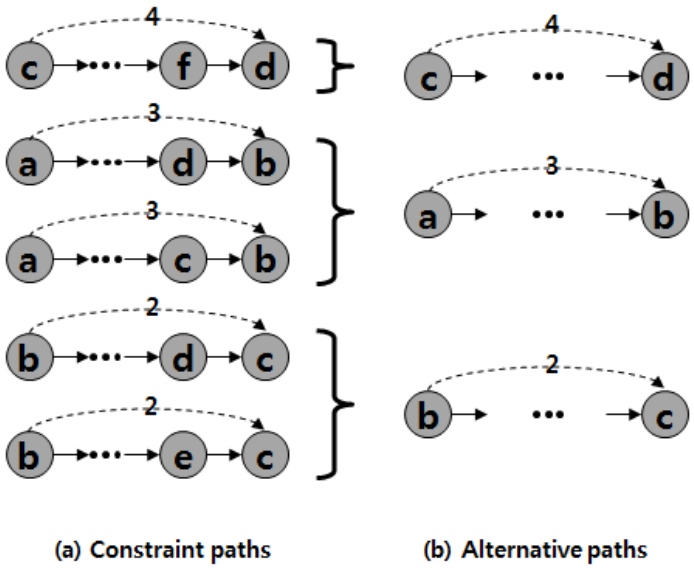


Figure 2. Construction of an Alternative Path Set from Constraint Paths

Figure 5(a) shows an example of construction of an alternative path set from the constraint paths. We find groups of constraint paths that have alternative paths. The number of alternative paths is m/k, where m is the number of constraint paths and k is the average number of constraint paths per each alternative path.

We give the complete algorithm of searching all the constraint segments. First, we create a view APT consisting of a set of alternative paths (line 1). For each alternative path p^a, we find all the constraint segments in a similar way with algorithm 1 (line 4-10).

Algorithm 2. Searching Constraint Segments with Optimization

Algorithm 2 SearchSegments(constraint path set S_{input})

•**Input**: constraint path set $S_{input} = \{p_1, p_2, \ldots, p_n\}$
•**Output**: $S_{output} = \{p | \forall p \in Index\ Table,\ S_{input} \cap p \neq \emptyset\}$

1: **CREATE** a temporary view APT having start node id, destination node id and distance using distinct alternative paths.
2: **FOR** each path p^a in APT
4: **IF** distance of p^a is longer than l_{thd}
5: **RETURN** ∅;
6: **EXPAND** node v using threshold $l_{thd} - cost(p^a)$
4: **CREATE** a temporary table FwdSegments and Insert all rows through line1 to FwdSegments, S
5: Similar actions from line 1 to line 2 for the backward expansion;
6. **FOR** each segment s_1 **IN** FwdSegments
7 **FOR** each segment s_2 **IN** BckSegments
8. **IF** cost(s_1) + cost(s_2) + $cost(p^a) \leq l_{thd}$
9. **Insert** $s_1 + s_2$ into S_{output}
10: **UPDATE** the cost of constraint segments as infinite
11. **RETURN** S

5. Experiments

In this section, we experimentally evaluate the effectiveness and the efficiency of our relational approach on a real dataset.

5.1. Experimental Setup

We implement all methods in Java with JDK 1.7 and evaluate them on 3.3GHz Intel® Core™ i7-3960X processor running Windows 7.

We use three kinds of real-world graph data sets including DBLP [20], GoogleWeb [21], and collaborative network [22]. Some statistics of these graphs are summarized in Table 1.

Table 1. Description of Data Sets

DataSet	# Nodes	# Edges	Edge label range
Collaborative network	40,421	175,691	[0.03, 9.9]
DBLP	312,967	1,149,663	1
GoogleWeb	875,713	5,105,039	1

In order to show the efficiency of our method, we conduct the following experiments over a commercial database system. For searching shortest path, we use the single directional set Dijkstra's approach.

(1) Comparison of the average time cost and the total space cost between the proposed method (ConstSegment) and a naïve approach that keeps all the edge information of each path segment in the index table (SegPath)

(2) Analysis of the time cost of the algorithm 1 according to the number of constraint segments

(3) Analysis of the time cost of partial SQL of algorithm 1

(4) Comparison of the average time cost between an iterative approach of algorithm 1 and algorithm 2

5.2. Performance Evaluation

The first experiment compares the execution time and space costs. We randomly generate 1000 shortest path queries with a constraint path. In Figure 6, the proposed method (ConstSegment) requires about total 30K rows and SegPath requires about 1,300K rows in CollaNet dataset. Likewise, ConstSegment requires about about 108K rows and SegPath requires about 5,000K rows in DBLP dataset. In the GoogleWeb dataset, ConstSegment needs about 33K rows and SegPath needs about 9,500K. Segpath stores edge information for all shortest segment, but the proposed method temporarily stores only constraint segments. Therefore, it shows that SegPath method requires 40 to 100 times more rows than the proposed method.

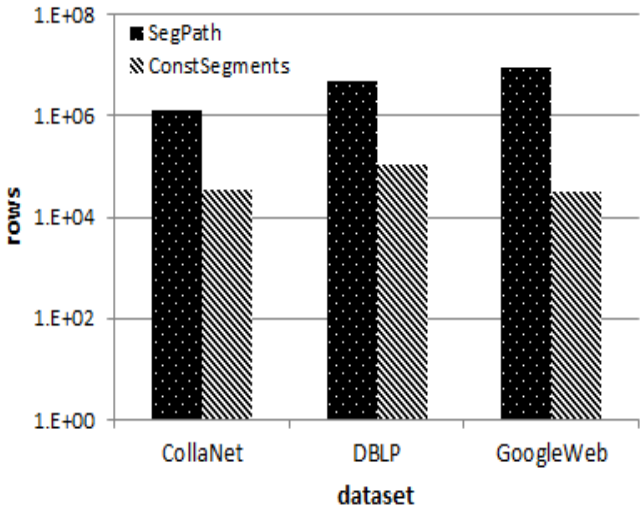


Figure 6. Comparisons of the Total Space Cost

In the Figure 7, the proposed method is improved the performance 10% to 50% in execution time cost compared with SegPath method. The proposed method is faster than SegPath because the SegPath requires additional operation that searches the constraint segments in the SegPath table. The proposed method can avoid constraint path from the original index table directly.

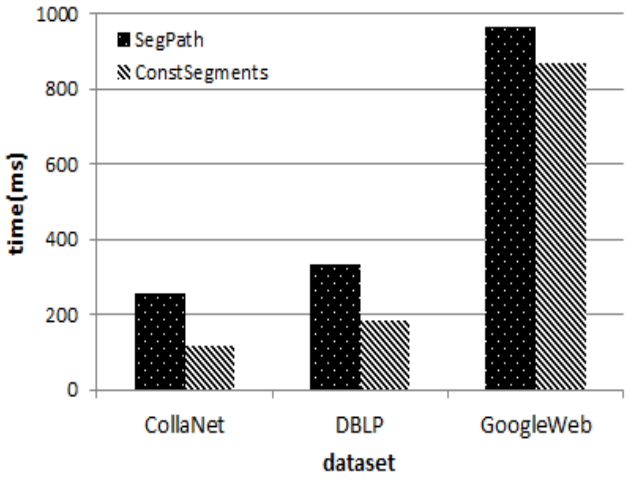


Figure 7. Comparisons of the Average Time Costs

The second experiment evaluates the average time cost of Algorithm 1 according to the number of constraint segments. In Figure8, the result shows almost similar execution time cost irrelevant to increase of the number of constraint segments in all datasets. The Algorithm 1 always guarantees two iterations irrelevant to the number of constraint segments.

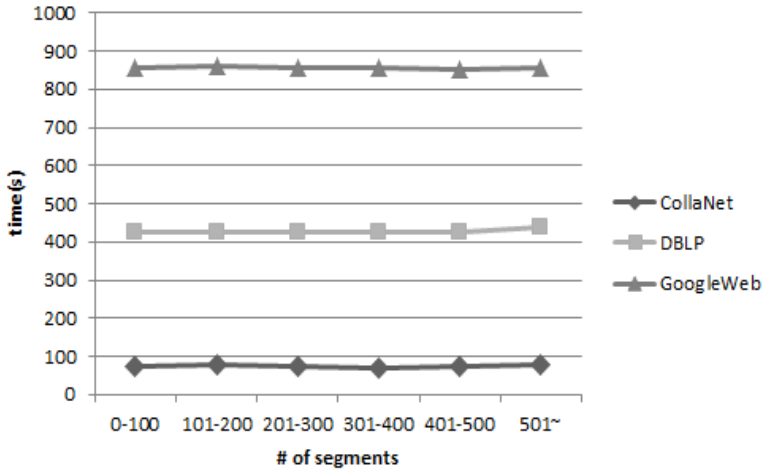


Figure 8. Comparisons of the Average Time Costs Varying the Number of Segments

The third experiment evaluates the time cost for each SQL statement according to parts of functions in Algorithm 1. We divide the algorithm into three parts. The first part is listing 1(1) for searching forward segments (fwd), the second part is listing 1(1) for searching backward segments (bck), and the third part is listing1(2-4) for merging forward and backward segments. We observe that fwd and bck occupy more than 80% of the total execution time. In order to construct fwd and bck, we need to scan the index table whose size is much larger than that of forward and backward segments.

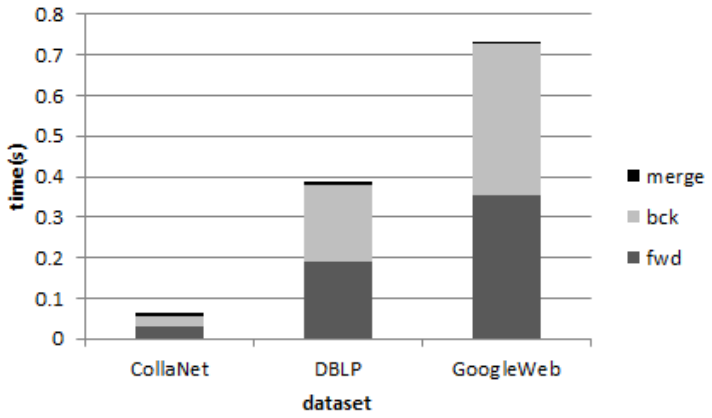


Figure 9. Comparisons of the Average Time Cost for each SQL in alg. 1

The fourth experiment compares algorithm 2 and iterative approach of algorithm 1. We set the parameter n as the average number of alternative paths for each constraint path. We randomly generate three sets of alternative paths that include 100 distinct alternative paths. In Figure 10, the time cost linearly increases in alg 1 according to increasing n. The proposed algorithm (alg 2) only requires about 1/ n execution time compared with that of alg 1. As we mentioned in the section 4.3, the result validates that the number of edge scanned is linearly decreasing according to n.

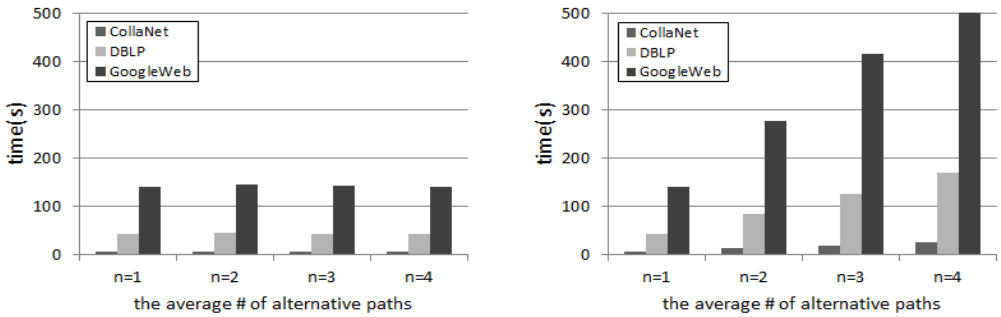


Figure 10. Comparisons of the Average Time Cost between alg. 2 and alg. 1

6. Conclusion

An RDB based efficient method has been proposed for shortest path searching considering a constraint path in a large-scale graph. The proposed method finds the shortest path not containing the constraint path by avoiding expanding the paths including the given constraint path. Experiments show the proposed method requires smaller disk space than a naïve method with similar execution time.

An RDB based efficient method has been proposed for shortest path searching considering a constraint path in a large-scale graph. The proposed method finds the shortest path not containing the constraint path by avoiding expanding the paths including the given constraint path. We also optimized the performance for multiple constraint paths by searching only once for the constraint paths having common expanding paths. Experimental results show the proposed method requires smaller disk space and time cost than the naïve method.

Acknowledgements

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2013R1A2A1A05056375).

References

[1] E. Dijkstra, "A note on two problems in connexion with graphs", Numerische Mathematik, vol. 1, no. 1, **(1959)**.
[2] D. Wagner and T. Willhal, "Speed-up techniques for shortest-path computations", Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science, Aachen, Germany, **(2007)** February 22-24.
[3] A. Goldberg and C. Harrelson, "Computing the shortest path: search meets graph theory", Proceedings of the 16th annual ACM-SIAM symposium on discrete algorithms SODA, Vancouver, British Columbia, **(2005)** January 23-25.
[4] F. Wei, "Tedi: efficient shortest path query answering on graphs", Proceedings of the 29th ACM SIGMOD International Conference on Management of Data, Indianapolis, USA, **(2010)** June 6-11.
[5] R. Prim, "Shortest connection networks and some generalizations", Bell System Technical Journal, vol. 36, no. 6, **(1957)**.
[6] D. Johnson and L. McGeoch, "The traveling salesman problem: A case study in local optimization", Local search in combinatorial optimization, **(1997)**.
[7] M. Ahmed and A. Lubiw, "Shortest Paths Avoiding Forbidden Subpaths", Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science, Freiburg, Germany, **(2009)** February 26-28.
[8] D. Villeneuve and G. Desaulniers. The shortest path problem with forbidden paths. European Journal of Operational Research, 165, 1 (2005)

[9] Y. Yuan, G. Wang, H. Wang and L. Chen. Efficient subgraph search over large uncertain graphs. PVLDB, 4, 11 (2011)
[10] D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. Discrete Applied Mathematics, 126, 1 (2003)
[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. Proceedings of the 6th symposium on Operating Systems Design & Implementation (2004), December 6-8, San Francisco, CA
[12] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized pagerank on mapreduce. Proceedings of the 30th ACM SIGMOD International Conference on Management of Data (2011), June 12-16, Athens, Greece
[13] S. Padmanabhan and S. Chakravarthy. HDB-Subdue: A Scalable Approach to Graph Mining. Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery (2009), August 31-September 2, Linz, Austria
[14] S. Chakravarthy and S. Pradhan. DB-FSG: An SQL-based approach for frequent subgraph mining. Proceedings of the 19th International Conference on Database and Expert Systems Applications, Turin, Italy, **(2008)** September 1-5.
[15] J. Gao, R. Jin, J. Zhou, J. Yu, X. Jiang and T. Wang, "Relational Approach for Shortest Path Discovery over Large Graphs", PVLDB, vol. 5, no. 4, **(2011)**.
[16] C. Wang, W. Wang, J. Pei, Y. Zhu and B. Shi, "Scalable mining of large disk-based graph databases", Proceedings of the 10th ACM SIGKDD international conference on knowledge discovery in databases, Seattle, WA, **(2004)** August 22-25.
[17] X. Yan and J. Han, "gSpan: graph-based substructure pattern mining", Proceedings of the 3rd International Conference on Data Mining, Florida, USA, **(2002)** November 19-22.
[18] L. Gouveia, P. Patr´ıcio, A. D. Sousa and R. Valadas, "MPLS over WDM network design with packet level QoS constraints based on ILP models", Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies, San Francisco, USA, **(2003)** March 30-April 3.
[19] K. Lee and M. A. Shayman, "Optical network design with optical constraints in IP/WDM networks", IEICE Transactions on Communications, vol. 88, no. 5, **(2005)**.
[20] J. Yang and J. Leskovec, "Defining and Evaluating Network Communities based on Ground-truth", Proceedings of the 18th ACM SIGKDD Workshop on Mining Data Semantics, Beijing, China, **(2012)** August 12-16.
[21] J. Leskovec, K. Lang, A. Dasgupta and M. Mahoney, "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters", Internet Mathematics, vol. 6, no. 1, **(2009)**.
[22] M. E. J. Newman, "The structure of scientific collaboration networks", National Academy of Sciences, vol. 98, no. 2, **(2001)**.

Authors

Jihye Hong, received her B.S. degree from Kyung Hee University, South Korea in 2012. She is Master candidate at Department of Computer Engineering, Kyung Hee University, South Korea since March 2012. Her research interests include large graph processing, graph mining, graph databases.

Yongkoo Han, received his B.S., M.S. and Ph. D. degree at Department of Computer Engineering at Kyung Hee University, South Korea in 2012. He was Postdoctoral Research Associate at Department of Computer Engineering, Kyung Hee University, South Korea since September 2012. He is a research professor in the Department of Computer engineering at Kyung Hee University, South Korea since 2013. His research interests including graph mining, Activity Recognition and Knowledge based System.

Young-Koo Lee, received his B.S., M.S. and Ph. D. in Computer Science from Korea advanced Institute of Science and Technology, South Korea in 2002. He was Postdoctoral Research Associate at Dept. of Computer Science, University of Illinois at Urbana-Champaign, Illinois, U.S.A. from Sept. 2002 to Feb. 2004. He is a professor in the Department of Computer Engineering at Kyung Hee University, South Korea since 2004. His research interests including graph mining, big data management and databases. He is a member of the IEEE, the IEEE Computer Society, and the ACM.