Comments on "Stack-based Algorithms for Pattern Matching on DAGs"

Qiang Zeng[1] Hai Zhuge[2]
Institute of Computing Technology, Chinese Academy of Sciences
Beijing 100190, China
[1]zengqiang@kg.ict.ac.cn [2]zhuge@ict.ac.cn

ABSTRACT

The paper "Stack-based Algorithms for Pattern Matching on DAGs" generalizes the classical holistic twig join algorithms and proposes PathStackD, TwigStackD and DagStackD to respectively evaluate path, twig and DAG pattern queries on directed acyclic graphs. In this paper, we investigate the major results of that paper, pointing out several discrepancies and proposing solutions to resolving them. We show that the original algorithms do not find particular types of query solutions that are common in practice. We also analyze the effect of an underlying assumption on the correctness of the algorithms and discuss the pre-filtering process that the original work proposes to prune redundant nodes. Our experimental study on both real and synthetic data substantiates our conclusions.

1. INTRODUCTION

Chen et al. [3] proposed a group of stack-based algorithms, namely PathStackD, TwigStackD and DagStackD, to evaluate path, twig and DAG pattern queries on directed acyclic graphs (DAGs). Several other studies (e.g. [7], [6], [4], [8]) have also implemented these algorithms for experimental comparison. This paper makes an in-depth analysis of PathStackD (Section 4) and TwigStackD (Section 5). We propose a classification on solutions to pattern queries and show that both PathStackD and TwigStackD do not find particular common types of solutions. We develop a modified algorithm for PathStackD and two modified algorithms for TwigStackD, and investigate the updated time and space complexity. In addition, we discuss two claims in [3] related to an underlying assumption about "optimum" tree-covers (Section 6) and a pre-filtering process (Section 7), showing several discrepancies in the original results. Finally, we present an experimental study substantiating our claims (Section 8).

2. DEFINITIONS, NOTATIONS, AND OPERATIONS

This section introduces some definitions, notations and operations used through the following discussion and analysis.

Pattern Matching Queries. A *path (twig/DAG) pattern query* is a directed path (resp. tree/DAG) $Q = (V_Q, E_Q, f_Q)$, where (1) V_Q is a set of finite nodes, and $E_Q = V_Q \times V_Q$ representing edges, (2) f_Q is a labeling function on V_Q such that for each node $q \in V_Q$, $f_Q(q)$ is a symbol in a finite alphabet, denoting a label associated with q. The *data graph* considered is a DAG $G = (V_G, E_G, f_G)$, where V_G, E_G, and f_G are respectively a finite set of nodes, a set of edges, and the labeling function. Let $|Q|$, $|V_G|$, $|E_G|$ denote the size of query nodes, the size of graph nodes and the number of graph edges, respectively. Given Q and G, assume $V_Q = \{q_1, \ldots, q_n\}$. A set of n data nodes $(t_{q_1}, \ldots, t_{q_n})$ in G is said to be a *solution* to Q (or a *match* of Q), if and only if the following conditions hold: (1) $f_Q(q_i) = f_G(t_{q_i})$, for $i \in [1, n]$; (2) For each edge $(q_i, q_j) \in E_Q$, t_{q_j} is a descendant of t_{q_i}, i.e., t_{q_j} is reachable from t_{q_i} in G. In this paper, we use t_q to denote a data node with the same label of a query node q and say t_q is a *matching node* of q. The answer to a pattern query Q is a set containing all solutions. As for a path (twig) query, a *descendant extension* of a data node t_q on G with respect to a child node q' of q is a solution to the subpath (resp. subtwig) rooted at q' that contains a descendant $t_{q'}$ of t_q. For example, (a_1, b_1, c_1) is a solution to the pattern query in the data graph shown in Figure 1. a_2 has a descendant extension w.r.t. B, (b_2, c_1), as it is a solution to the path B–C and b_2 is a descendant of a_2. We define *ancestor extensions* of a data node analogously.

In this paper, a spanning tree of a data graph G is also called a tree-cover [1]. Given a data graph G and a tree-cover (TC), a data node t_{q_i} is said to be a *TC descendant* of another data node t_{q_j} and t_{q_j} is said to be a *TC ancestor* of t_{q_i}, if t_{q_i} is a descendant of t_{q_j} in TC; otherwise if t_{q_i} is reachable from t_{q_j} in G but not in TC, t_{q_i} is said to be an *NTC descendant* of t_{q_j} and t_{q_j} is an *NTC ancestor* of t_{q_i}. We next define four types of solutions to a pattern query.

Classification of Solutions. Given a (path/twig/DAG) query Q and a graph G with a TC, assume $(t_{q_1}, \ldots, t_{q_n})$ is a solution to Q. We call an edge $(q_i, q_j) \in E_Q$ a TC edge *with respect to* that solution if t_{q_j} is a TC descendant of t_{q_i}; otherwise we call it an NTC edge. A solution is of *Type-1* if all edges in Q are TC edges; it is a *Type-2* solution if (1) there are at least one TC edge and one NTC edge w.r.t. the solution, and (2) for every NTC edge $(q_i, q_j) \in E_Q$, all edges directing to the ancestors of q_i (if any) are also NTC edges; it is a *Type-3* solution if all edges in Q are NTC edges w.r.t. it; it is a *Type-4* solution if (1) there are at least one TC edge and one NTC edge with respect to the solution, and (2) there is an NTC edge $(q_i, q_j) \in E_Q$ such that at least one edge directing to an ancestor of q_i is a TC edge. According to the definition, Type-2 and Type-4 solutions constitute of multiple Type-1 and Type-3 subsolutions. Clearly, the four types of solutions are disjoint and include all kinds of possible solutions. For example, in Figure 1, $(a_2, b_2, c_2), (a_3, b_2, c_2), (a_3, b_2, c_1)$, and (a_1, b_1, c_1) are respectively a Type-1, Type-2, Type-3 and Type-4 solution to the path query on the graph. Type-2, Type-3 and Type-4 solutions

Proceedings of the VLDB Endowment, Vol. 5, No. 7

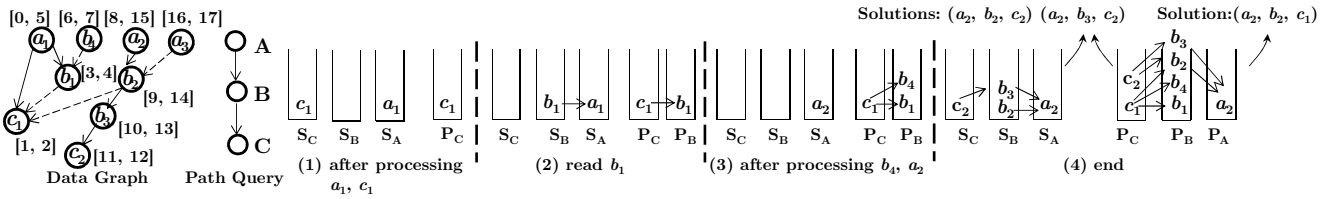


Figure 1: Snapshots of stacks and pools during the evaluation of PathStackD. In the data graph, the solid edges represent the edges in the tree-cover, and the dashed edges represent non-tree edges.

are collectively referred to as *NTC solutions*, while Type-1 solutions are also called *TC solutions*. Similarly, we classify descendant extensions of a data node to TC ones and NTC ones. Given a descendant extension of t_q w.r.t. a child node q' of q, if all edges in the corresponding subquery are TC edges w.r.t. the extension and $t_{q'}$ is a TC descendant of t_q, it is called a *TC descendant extension*; otherwise, it is an *NTC descendant extension*. In Figure 1, (b_2, c_2) is a TC descendant extension of a_2 w.r.t. B, while (b_2, c_2) and (b_2, c_1) are two NTC ones of a_3.

Streams, Stacks, and Pools. [3] extends the holistic twig join algorithms, namely PathStack and TwigStack [2] that evaluate path and twig queries over trees, to process queries on DAGs by utilizing a reachability index and incorporating a data structure called *partial solution pool* (essentially a list). Each query node q is associated with a stream T_q, a stack S_q and a pool P_q. The stream T_q contains all data nodes having the same label as associated with q (i.e. a query variable binding). During the evaluation process, the set of stacks encodes the partial and total solutions to the query on TC, and the pools encode the partial and total solutions to the query beyond TC but on the whole G. T_{q_1} is said to be a *parent stream* of T_{q_2}, if q_1 is a parent of q_2, that is, $(q_1, q_2) \in E_Q$. Similarly, we define *parent/child stacks* , *parent/child pools* according to the parent-child relationship between the corresponding query nodes. We use $|T|_{total}$ to denote the total number of stream nodes, and $|T|_{max}$ to denote the maximal size of a stream.

Operations. The basic operations on streams are `eof`, `advance`, `next`. Initially, a pointer is attached to each stream and point to the position *before* the first node. `eof`(T_q) tests whether the pointer has come to the end of the stream T_q. `advance`(T_q) moves the pointer to the next node. `next`(T_q) return the node next to the pointer (note that the pointer is not moved). Operations `push`, `pop`, `top` on stacks respectively push a node, pop and return the top node. As for nodes in a path or a twig, `subtree(`q`)` returns *all* nodes in the subtree rooted at q, and `parent(`q`)` and `children(`q`)` return the parent and all children of q respectively.

3. REVIEW

This section gives an overall review of [3], presenting its main results and discussing the algorithms through examples.

Interval+SSPI. As answering reachability queries is a building block for evaluating pattern matching queries, [3] proposes an reachability index scheme, called Interval+SSPI in this paper, by combining the classical interval encoding on a tree-cover of a graph and a predecessor index, namely surrogate & surplus predecessor index (SSPI). Each data node v is associated with an interval $[v.start, v.end]$, where $v.start$ and $v.end$ are the pre-order number and the post-order number in a graph traversal. The reachability between two nodes in TC generated by the graph traversal can be directly determined by checking the containment relationship between the intervals of the two nodes in $O(1)$ time. In addition to the interval encoding, a predecessor list, denoted by $PL[v]$, is assigned to each node v to derive the remaining transitive closure not covered by the tree-cover. Given two nodes v_1 and v_2, if v_1 can reach v_2 and the interval of v_1 does not contain that of v_2, the path from v_1 to v_2 must contain at least an edge not in TC (called a non-tree edge). Suppose (v_1', v_2') is the last non-tree edge (i.e. closest to v_2) on the path. If $v_2' \neq v_2$, v_2' is put into $PL[v_2]$ and called a surrogate predecessor of v_2; otherwise, v_1' is put into $PL[v_2]$ and called a immediate surplus predecessor of v_2. By recursively looking up nodes in the predecessor lists as intermediate nodes and checking the containment of intervals, a reachability query can be answered in $O(|E_G|)$ time. The total size of predecessor lists is bounded by $(|E_G| - 1)$. For more details of query processing, please refer to [3]. Take the data graph of Figure 1 as an example, where $PL[c_2] = \{b_2\}$ and $PL[b_2] = \{a_3\}$. The reachability query between c_2 and a_3 can be answered by the following steps: (1) first look up b_2 in $PL[c_2]$, and examine whether $[b_2.start, b_2.end]$ is subsumed by $[a_3.start, c_3.end]$; (2) since the interval [9, 14] of b_2 is not contained in the interval [16, 17] of a_3, continue to look up $PL[b_2]$; and (3) a_3 is found in $PL[b_2]$, so we conclude that a_3 can reach c_2.

PathStackD. The original algorithm PathStackD is presented in Algorithm 1. PathStackD visits stream nodes in the ascending order of their $start$ values (line 2). For each node $t_{q_{min}}$, the algorithm pops nodes from stacks that are not TC ancestors of $t_{q_{min}}$ (line 4–6) and pushes $t_{q_{min}}$ into $S_{q_{min}}$, adding a pointer to $t_{q_{min}}$ pointing to the top node in the parent stack (line 8). PathStackD expands the NTC descendant extensions of $t_{q_{min}}$ stored in pools in line 6. When q_{min} is a leaf node, the partial or total solutions are transferred to pools (line 32) and next output if they are total solutions (line 33); when q_{min} is a root node, the total solutions headed by $t_{q_{min}}$ are output from pools after the descendant extensions are expanded (line 29–30). The evaluation is terminated when all leaf stream nodes are processed.

The key difference between PathStackD and PathStack is that PathStackD introduces a pool structure. A node t_q is saved in P_q, if q is a leaf node or t_q has a descendant extension in child pools or stacks. Indeed, stacks are only able to encode solutions to a path query on trees, so NTC solutions need to be stored using a structure other than stacks. Due to the rule of the pop operations on stacks, nodes in a stack are clearly in a leaf-to-root order from top to bottom. A node has exactly one pointer to an ancestor in the parent stack. In contrast, nodes in a pool (1) do not follow a leaf-to-root order, (2) cannot be dynamically removed from the pool, and (3) may have more than one pointers to the parent pool.

Figure 1 shows an example. PathStackD first processes a_1, c_1, b_1, and b_4 and pushes them into stacks in order. a_1 is not put in P_A, because when a_1 is retrieved from T_A, P_B is empty and obviously a_1 cannot reach any in P_B. Because C is a leaf node, c_1 becomes the first node being put in the pool structure, followed by b_1 due to that b_1 is able to find c_1 in its child pool, which forms a descendant extension. When b_4 is pushed into S_B, both b_1 and a_1 are popped from the stacks. PathStackD then attempts to find NTC descendant extensions of a_2 in pools. Because a_2 cannot reach

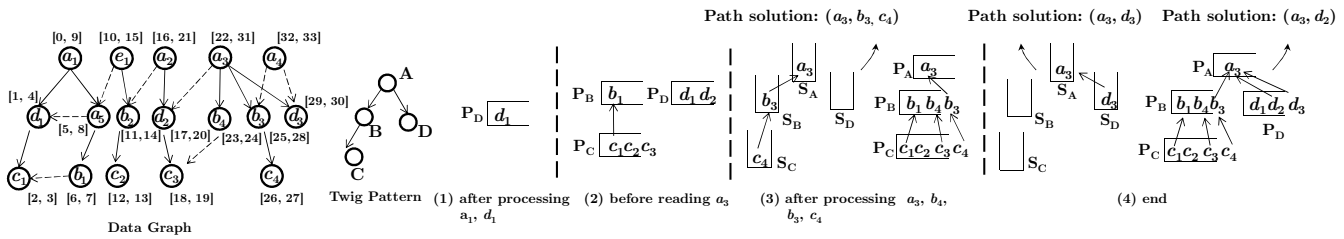


Figure 2: Snapshots of stacks and pools during the evaluation of TwigStackD

b_1–the only node in the child pool P_B of P_A, a_2 is not put into P_A; but it will eventually be moved to the pool when a TC solution (a_2, b_3, c_2) is found in stacks. After processing c_2, the evaluation process is terminated, leaving a_3 not processed. In the end, two solutions (a_2, b_2, c_2) and (a_2, b_3, c_2) are encoded in the stacks, since c_2 points to b_3, b_3 points to a_2 and b_2 is below b_3 in S_B. Besides, another solution (a_2, b_2, c_1) can be obtained from the pools.

We can draw the following conclusions from this example. (1) The two solutions obtained from the stacks are exactly the answer to the path query on the tree-cover, that is, Type-1 solutions. (2) The pools encode duplicate solutions with stacks. Indeed, in procedure `showSolutions`, all stack solutions are moved to the pools, resulting in redundancy. (3) There is a redundant intermediate solution (b_4, c_1) stored in pools that does not contribute to any solution to the path query. (4) The evaluation gets a Type-4 solution (a_2, b_2, c_1) from the pools, but fails to find another Type-4 solution (a_1, b_1, c_1), one Type-3 solution (a_3, b_2, c_1), and two Type-2 solutions (a_3, b_2, c_2) and (a_3, b_3, c_2). The last three solutions are not identified because the evaluation is terminated too early.

TwigStackD. TwigStackD (Algorithm 3) consists of two stages. In the first stage, it constructs the stack encoding of partial/total TC path solutions and the pool encoding of partial/total NTC path solutions by processing nodes in the ascending order of $start$ values. In the second stage, all path solutions are merged to derive the final answer to the twig pattern query. Specifically, for each stream node t_q to be processed, TwigStackD first identifies the "missing" child query nodes of q such that t_q has no TC descendant extensions w.r.t. them (line 5). It then determines whether an NTC descendant extension exists in pools w.r.t. each "missing" node q', by checking the ancestor-descendant relationship between t_q and nodes in $P_{q'}$ (line 6). If t_q have descendant extensions w.r.t. all child query nodes, it is added into P_q to expand those extensions in pools. Note that the operation in line 24 is not in the original algorithm but actually needed, because for a leaf stream node, there is of course no "missing" child query nodes and should be put into the pool anyway. Since the detail of function `showSolutionsWithBlockings` is not given in [3], we assume that similar to `showSolutions` in PathStackD, the solutions in stacks are replicated to the pools. After all nodes in the leaf streams are processed, all individual path solutions are merged to form twig solutions.

Figure 2 illustrates an evaluation process of TwigStackD. Because a_1 has no TC descendant extension w.r.t. B and also certainly cannot find one in the empty pools, a_1 is not pushed into S_A. d_1 is either not pushed into S_D due to the lack of an ancestor in S_A; but it is put into P_D, because it is a match of the leaf node D. a_3 is the first node pushed into the stack structure, because it directly finds two TC descendant extensions respectively w.r.t. B and D. In the end, we obtain two path solutions encoded in the stacks and one new path solutions in pools. By merging the path solutions, TwigStackD only finds one Type-1 twig solution (a_3, b_3, c_4, d_3) and one Type-2 solution (a_3, b_3, c_4, d_2).

The following conclusions about this example can be made. (1) TwigStackD correctly finds the Type-1 solution. Indeed, the correctness of TwigStack guarantees TwigStackD is able to identify all twig solutions on a tree. (2) Similar to PathStackD, pools encode duplicate solutions with stacks and redundant intermediate solutions. (3) TwigStackD fails to find two Type-4 solutions (a_1, b_1, c_1, d_1) and (a_5, b_1, c_1, d_1), and two Type-2 solutions (a_2, b_2, c_2, d_2) and (a_4, b_3, c_4, d_3). The last solution is not found because of the early termination of the first stage of the evaluation.

DagStackD. [3] briefly introduces another algorithm DagStackD for DAG pattern queries. DagStackD first decomposes a DAG pattern into several twig patterns, evaluates them using TwigStackD individually, and merges the results of each twig query to assemble the final answer. Due to space limitations, [3] does not elaborate on DagStackD. Since TwigStackD servers as a sub-routine, we can infer that DagStackD suffers from the same problems as TwigStackD.

We investigate PathStackD and TwigStackD in Section 4 and Section 5 respectively. We assume that the tree-cover of a data graph and the intervals are created by a depth-first graph traversal so that the order of the end values of the intervals is exactly reverse topological order. We shall relax this constraint and discuss its effect in Section 6.

4. ANALYSIS OF PATHSTACKD

PathStackD generalizes PathStack to evaluate path pattern queries on DAGs. As our example shows in Section 3, an obvious problem with the original PathStackD is the termination time. Since it is a less fundamental issue, *we assume in the following discussion of the capability of PathStackD that this problem has been fixed (the solution is given when we present our modified version) and all necessary stream nodes can be processed.*

Observe that all Type-1 solutions (TC solutions) can be correctly found by PathStackD, due to the correctness and completeness of PathStack. The key challenge of PathStackD is how to encode NTC solutions in pools. A node t_q may be added into P_q in two situations. The first is when t_q has the minimal $start$ value in stacks and has a descendant already in $P_{child(q)}$ so that a descendant extension in pools can be expanded (line 18 of Algorithm 1). The other is when PathStackD identifies a solution containing t_q encoded in stacks and transfers it into pools (line 32 of Algorithm 1).

We first consider the first situation. Note that (1) the nodes are added to pools in reverse topological order, and (2) no NTC descendant extensions of a node t_q would be created and expanded once after t_q is put into P_q. Accordingly, to ensure all NTC solutions can be derived from pools, PathStackD should guarantee that, before determining whether a node should be put in its pool, all of its NTC descendant extensions have been already stored in pools. However, PathStackD does not satisfy this requirement. The cause is that PathStackD processes stream nodes in the pre-order rather than the reverse topological order. One intuitive observation about this is that the solutions in stacks are expanded from the root to leaf, while those in pools are expanded from the leaf to root.

Algorithm 1: PathStackD [3]

```
1.  while ¬ end() do
2.      q_min := argmin_i{next(T_qi).start}
3.      t_qmin = next(T_qmin)
4.      for each q_i in subtree(q) do
5.          while ¬empty(S_qi) ∧ top(S_qi.end) < t_qmin.start do
6.              pop(S_qi)
7.      sweepPartialSolutions(q_min)
8.      push(S_qmin, t_qmin, pointer to top(S_parent(q_min)))
9.      advance(T_qmin)
10.     if isLeaf(q_min) then
11.         showSolutions(S_qmin, 1, null)
12.         pop(S_qmin)

    Function: end()
13. for each leaf query node q_i do
14.     if ¬eof(T_qi) then return false
15. return true

    Procedure: sweepPartialSoltuions(q)
16. t_q := next(T_q)
17. for each h in P_child(q) do
18.     if checkContainment(t_q, h) then expand(q, tq, h)

    Function: checkContainment(t_q, h)
19. found := false
20. while ¬empty(PL[h]) ∧ ¬found do
21.     a := first(PL(h))
22.     if a.start > t_q.start ∧ a.end < t_q.end then return true
23.     else if a.start > t_q.end then return false
24.     else if ¬(found := checkContainment(t_q, a)) then
25.         PL[h] := PL[h]\{a} ∪ PL[a]
26. return found

    Procedure: expand(q, t_q, h)
27. put t_q into P_q
28. h.ptr_to_parentPool := t_q
29. if isRoot(q) then
30.     output the solutions headed by t_q in the root pool

    Procedure: showSolutions(SN, TC, ChildSP)
31. index[SN] := TC
32. expand(SN, S[SN].index[SN], S[SN + 1].ChildSP)
33. if SN = 1 then output(S[1].index[1], ..., S[n].index[n])
34. else
35.     for i = 1 to S[SN].index[SN].ptr_to_stackParent do
36.         showSolutions(SN − 1, i, TC)
```

Recall the example shown in Figure 1. (b_1, c_1) is an NTC descendant extension of a_1 and cannot be identified by stack encoding; but both b_1 and c_1 are TC descendant of a_1 and hence processed after a_1. That is, when PathStackD expands possible partial solutions in pools with a_1, b_1 and c_1 are still not in the pools. As a result, a_1 is seen as not having descendant extensions w.r.t. B and consequently PathStackD is not able to find the Type-4 solution (a_1, b_1, c_1). Interestingly, although PathStackD processes nodes in pre-order, those nodes having no ancestor-descendant relationship in the tree-cover are read in reverse topological order, since it is assumed that the intervals of nodes are produced by a depth first traversal. Therefore, we have the following critical Observation 1. Since Type-3 solutions have no Type-1 parts, the observation shows that PathStackD can correctly find all of Type-3 solutions.

Observation 1. *A solution partially containing Type-3 parts can be identified by PathStackD as long as the Type-1 parts (if any) can be constructed and expanded completely and correctly.*

We now turn to the second situation of nodes being put in pools. We have the following observations: (1) the second situation is the only chance for a Type-1 partial solutions to be encoded in pools; (2) no partial solutions in pools would be expanded when a Type-1 partial/total solution is moved to the pools, unless a part of the transferred Type-1 solution "accidentally" connects with them. In the example of Figure 1, (a_2, b_2, c_1) can be derived from the pools, because the Type-1 part, i.e. (a_2, b_2), belongs to a Type-1 solution (a_2, b_2, c_2) and (b_2, c_1) is connected to a_2 when the Type-1 solution is replicated from stacks to pools. Therefore, for any solutions containing Type-1 partial solutions, they can be obtained by PathStackD, only if the Type-1 parts can be encoded in stacks and moved to pools, a case that only takes place in `showSolutions` and when a leaf matching node is found. In other words, we have Observation 2, showing what Type-2 and Type-4 solutions can be found.

Observation 2. *Type-2 and Type-4 solutions can be found only if each Type-1 part (partially) constitutes one TC ancestor extension of a leaf matching node.*

Clearly, the Type-1 part of a Type-2 solution satisfies the condition, but the parts of a Type-4 solution do not necessarily satisfy it. By Observation 1 and Observation 2, we have the following proposition.

Proposition 1. *Given a data graph and a path pattern query, all Type-1, Type-2 and Type-3 solutions can be found by PathStackD, but Type-4 solutions can be found if and only if each Type-1 part of them (partially) constitutes one TC ancestor extension of a leaf matching node.*

The problem for finding all Type-4 solutions is mainly how to resolve the conflict between the two orders for constructing stack encoding and pool encoding. A naïve solution is to simply put all stream nodes in their corresponding pools before any operations in the original PathStackD. However, it would introduce large redundant intermediate solutions in pools and incur high I/O cost. It is desirable to maintain the following rule to reduce the size of intermediate results and improve I/O efficiency.

Rule 1. *A node t_q is put into P_q if and only if it has at least one NTC descendant extension w.r.t. the child of q (if exists).*

In fact, the original algorithm is intended to follow Rule 1, but it does not correctly find all potential NTC descendant extensions and only maintains the property that all nodes being put in the pools have at least one descendant extension.

Our modified parts of PathStackD are presented in Algorithm 2. First of all, the evaluation terminates only if the root stack is empty and all nodes in the root stream have been read, since no new solutions will then be encoded in stacks and no descendant extensions will be possibly expanded to a total solution (line 3).

The most important modification is in procedure `expand` for expanding and finding Type-4 solutions (line 7–13). As illustrated in Figure 3, a Type-4 solution consists of Type-1 and Type-3 subpath solutions alternated with each other. When PathStackD identifies a descendant t_{q_k} in P_{q_k} for t_{q_j} , besides t_{q_j} itself, the ancestors of t_{q_j} in stacks, such as t_{q_i}, also should be put into the pools immediately, so that the NTC descendant extension currently rooted at t_{q_k} can be expanded with the whole Type-1 subpath solution from t_{q_i} to t_{q_j}, regardless of whether the subpath is part of a Type-1 total solution. In order to make the expansion possible, the order of pushing a node into its stack and sweeping pool nodes should be interchanged so that the ancestor extensions stored in stacks can be

Algorithm 2: Modified PathStackD

Modifications: (1) replace line 1 in Algorithm 1 with line 1 below, insert line 3 below to the position preceding line 7 in Algorithm 1 and interchange line 7 and line 8 in Algorithm 1 as shown below; (2) replace line 32 in Algorithm 1 with line 15-16 below; (3) replace procedure `expand` in Algorithm 1 with the one below.

Procedure: main
1. **while** $true$ **do**
2. $\quad\cdots$
3. $\quad$ **if** empty$(S_{q_{root}}) \wedge eof(T_{q_{root}})$ **then** break
4. $\quad$ push$\big(S_{q_{min}}, t_{q_{min}}$, pointer to top$(S_{\text{parent}(q_{min})})\big)$
5. $\quad$ sweepPartialSolutions(q_{min})
6. $\quad\cdots$

Procedure: expand(q, t_q, h)
7. **if** $\neg isInPool[t_q]$ **then** $P_q := P_q \cup \{t_q\}$
8. $t_q.ptrs\_to\_childPool := t_q.ptrs\_to\_childPool \cup \{h\}$
9. **if** $\neg isInPool[t_q]$ **then**
10. $\quad$ **if** $\neg$isRoot$(q) \wedge \neg$empty$(S_{parent(q)})$ **then**
11. $\quad\quad$ **for** $i := 1$ to $t_q.ptr\_to\_stackParent$ **do**
12. $\quad\quad\quad$ expand(parent(q), $S_{\text{parent}(q)}[i]$, t_q)
13. $\quad isInPool[t_q] := true$

Procedure: showSolutions$(SN, TC, ChildSP)$
14. $\cdots$
15. **if** $SN \neq 1$ **then**
16. $\quad$ expand$(SN, S[SN].index[SN], S[SN+1].ChildSP)$
17. $\cdots$

directly derived during expanding (line 4–5). In this way, all and only qualified nodes are able to be added into pools. Consequently, all Type-4 solutions can be correctly constructed and Rule 1 also follows.

Besides, we make the following modifications. (1) The pointers assigned to a pool node now points to the child pool instead of the parent pool, because solutions typically require nodes sorted in root-to-leaf order (note that the pointers in stacks, however, must point to the child stack due to the special scheme of stack encoding). (2) Since a node may have multiple descendants in the child pool and the connectivity should be individually specified, a list of pointers are attached to each pool node. In the original algorithm, $ptr\_to\_parentPool$ is actually also a list but not explicitly expressed in [3]. (3) $isInPool$ records for each node whether it is in a pool to determine whether the ancestor extensions in stacks have been already put in pools. (4) An `if` clause is added in procedure `showSolutions` to avoid pools storing duplicate TC solutions in stacks so that solutions derived from pools and stacks are distinctive (line 15–16).

Figure 4 shows the updated evaluate process of the modified PathStackD for the query in Figure 1. (1) The Type-4 solution (a_1, b_1, c_1) is correctly returned by the algorithm. Indeed, when b_1 is pushed into P_B, the partial solution (b_1, a_1) in stacks is also put into the pools and expanded with c_1. (2) a_2 is not added into the pool, since a) when it is processed, no descendants are found in P_B and it has no descendant extensions in pools; b) when performing procedure `showSolutions`, only partial solutions (b_2, c_2) and (b_3, c_2) rather than total solutions are transferred into pools. Consequently, the Type-1 solutions are not stored in pools. (3) The evaluation completes after processing a_3. All the NTC descendant extensions of a_3 can thus be expanded to the total solutions.

Time Complexity. We first point out three discrepancies in [3] concerning the analysis of the time complexity of the original algorithm which are independent of our extensions.

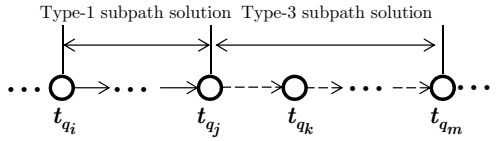


Figure 3: Illustration of a Type-4 path solution. The solution is embedded in the path pattern. The dashed edges represent the NTC ancestor-descendant relationship between two incident nodes, while the solid edges represent the TC ancestor-descendant relationship.

First, [3] only considers the cost for constructing stack and pool encoding, and does not take into account the cost of moving partial solutions from stacks to pools (line 32 of Algorithm 1) and the cost of outputting the final results. The size of partial and final solutions is bounded by $O(|T|^{|Q|}_{max})$, and so is the cost.

Second, the original analysis of the number of nodes being looked up in SSPI during an evaluation, on which [3] mainly focus, is also questionable. When determining the reachability between two nodes using Interval+SSPI, function `checkContainment` dynamically removes a part of the predecessors in the predecessor lists to reduce the times for recursively invoking `checkContainment`. [3] then claims that the total number of the removed nodes would not be larger than the size of the original SSPI, and gives an analysis on the basis of this claim. However, function `checkContainment` not only removes predecessors from SSPI, but also augments the index; thus the size of SSPI does not necessarily monotonously decrease. See line 24-25 in Algorithm 1. When the predecessor a in $PL[h]$ is not reachable from t_q, a is removed from $PL[h]$, and nodes in $PL[a]$ are replicated into $PL[h]$ but cannot be cleared from $PL[a]$. Therefore, if no nodes in $PL[a]$ are removed after PathStackD invokes `checkContainment` in line 24 (e.g. when all nodes in $PL[a]$ are to the right of t_q), the total size of predecessors in $PL[a]$ and $PL[h]$ is increased by $(|PL[a]| - 1)$. Indeed, during the evaluation, the size of SSPI may be even larger than $(|E_G| - 1)$, because the index nodes are propagated through the graph and the propagation does not keep the way of constructing SSPI that guarantees the bound of the index size.

Although the total size of SSPI is not bounded by $(|E_G| - 1)$ as a result of the add-and-remove operation in `checkContainment`, as for a single node h, the maximal size of predecessors to be looked up for checking the reachability to h is bounded by $(|E_G| - 1)$, since once a is removed, a will not be looked up again while nodes in $PL[a]$, which were potential predecessors to be looked up, are now replicated to $Pl[h]$ and may also have to be checked for the next reachability checking with h. Given a node h, we use ts_h, ss_h and nr_h to respectively denote the total size of nodes being looked up for all sweeping processes on h, the shrinking size of nodes, and the number of times when a node is looked up but not removed. Because h is not swept by duplicate nodes, we have $ts_h = ss_h + nr_h$ and $ss_h \leq |E_G| - 1$. In a call of `checkContainment`, there are two types of predecessors that are looked up in PL and not removed: the first is those to the right of t_q, and the second is those being looked up in the last recursive call stack (when $found = true$ in line 24 of Algorithm 1). During a recursion of `checkContainment`, whenever a recursive call in the depth higher than one encounters a node $a' \in PL[a]$ of the first type, a will be removed in a predecessor list in a call of lower depth. That is, at least a node would be removed if there is a node of the first type in a call with depth higher than one. Therefore, we have $ss_h \geq nr_h + 1 + |dph|_{max}|invoke|$, where $|dph|$ is the length of the longest recursive call stack and $|invoke|$ is the number of

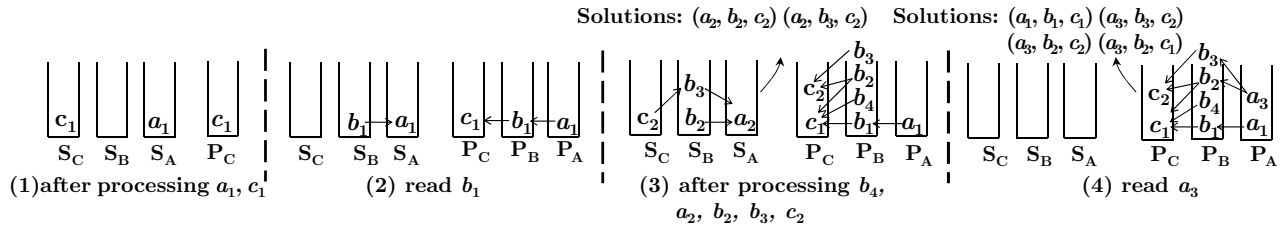


Figure 4: Snapshots during the evaluation of the modified PathStackD on the query and the data graph in Figure 1

times for h being checked by nodes in `sweepPartialSolutions` for examining reachability. In fact, in `checkContainment`, all predecessors that are looked up and to the right of t_q including those in the last linear call stack can be safely removed and its predecessor lists are subsequently duplicated. With this change, we can refine the previous expression to $ss_h \geq nr_h + 1$. Thus, $ts_h = ss_h + nr_h \leq 2ss_v - 1 \leq 2|E_G| - 3$. Given that there are $|T|_{total}$ stream nodes that may be all put in pools and need to be swept by others to the right in the worst case, the total times of predecessors in SSPI being looked up is $O(|E_G||T|_{total})$.

Third, the add-and-remove operations in `checkConainment` not only possibly increase the total size of index, but also involve cost for maintaining the order in the predecessor lists. The nodes in a predecessor list are initially sorted in ascending order of $start$ values. Because merging two sorted lists takes linear time in the sum of the sizes of two lists and the size of a predecessor list is bounded by $|V_G|$, each replication from a predecessor list to another when a predecessor is removed is in $O(|V_G|)$ time. In the worst case, such a replication should be performed whenever there is a node to be deleted. As discussed before, the shrinking size is $O(|E_G||T|_{total})$. Hence the total cost for maintaining the order in SSPI is $O(|E_G||V_G||T|_{total})$.

Plus the cost for stack encoding which is linear in $|V_G|$, the overall time complexity of the original PathStackD is $O(|T|_{total}|E_G|$ $|V_G|+|T|^{|Q|}_{max})$. Compared with the original PathStackD, the modified PathStackD needs to enumerate more intermediate solutions encoded in stacks and store them in pools. However, since the size of the intermediate solutions is bounded by $O(|T|^{|Q|}_{max})$, the complexity of the modified version does not change. Clearly, the time complexity of PathStackD is not independent of the size of intermediate solutions.

Space Complexity. The space cost for PathStackD includes the cost for stacks and pools. The space cost for stacks is $\min\{|T|_{total}, path_{max}\}$, where $path_{max}$ is the maximum size of a path in the tree-cover, since every state of stacks at a time corresponds to a path in the tree-cover. The major concern here is the space cost of pools. In the worst case, each stream node is put in its pool and every node has pointers to each ancestor in the parent pool. Hence the space complexity is $O(|T|^2_{max}|Q| + \min\{|T|_{total}, path_{max}\})$. The modified PathStackD does not increase the space complexity. Similar to the time complexity, the space complexity of PathStackD is not independent of the size of intermediate solutions. Indeed, the nodes in pools are only guaranteed to have descendant extensions but do not necessarily have ancestor extensions. The pools probably encode a large number of redundant intermediate results that do not contribute to any final solutions.

5. ANALYSIS OF TWIGSTACKD

Recall that TwigStackD finds solutions to individual paths in the first stage and then joins them to form final solutions in the second stage. The version in [3] has a problem with the termination time of the first stage that can be resolved the same as for PathStackD. We again do not consider it in our discussion until giving our modifications.

TwigStackD uses stacks and pools to construct TC and NTC path solutions respectively. The two following rules are for determining whether a node should be pushed in its stack and put in its pool. Note that Rule 3 does not lead to the conclusion that nodes having required descendant extensions are in pools, unless all possible descendant extensions can be found and expanded correctly.

Rule 2. *A node t_q is pushed in S_q only if it is in a TC solution to the twig query.*

Rule 3. *A node t_q is put in P_q only if TwigStackD finds the descendant extensions of t_q w.r.t. each of children(q).*

We have the following observations: (1) All Type-1 and Type-3 solutions can be correctly found by TwigStackD. (2) Type-2 and Type-4 solutions can be found as long as each Type-1 parts can be constructed and expanded completely and correctly. (3) For any Type-2 or Type-4 solutions that can be found, every Type-1 part must partially constitute a total Type-1 solution. The first two points are based on the analysis similar to that for PathStackD. The third point is due to Rule 2. Type-1 partial solutions, as a component of a Type-2 solution and a Type-4 solution, are first constructed in stacks only if the partial solutions participate in a total solution. Because only the Type-1 partial solutions that are encoded in stacks are moved to the pools and able to be expanded to total Type-2 or Type-4 solutions in `sweepPartialSolutionsTSD`, the Type-2 and Type-4 solutions having partial solutions that belong to Type-1 but do not take part in any final Type-1 solutions cannot be constructed and found in pools. In the example shown in Figure 2, TwigStackD fails to find the Type-2 solution (a_2, b_2, c_2, d_2), because the Type-1 partial solution (b_2, c_2) does not appear in any Type-1 total solution and hence cannot be constructed in pools when TwigStackD attempts to find descendant extensions for a_2. In contrast, if a_4 is processed with correct termination time, the Type-2 solution (a_4, b_3, c_4, d_3) could be found since (b_3, c_4) and (d_3) then have been moved from stacks to pools.

With the three observations, we can consequently prove Proposition 2. An obvious corollary is that TwigStackD cannot find any Type-2 nor Type-4 solutions if there are no Type-1 solutions.

Proposition 2. *Given a data graph and a twig pattern query, a Type-2 or Type-4 solution can be found by TwigStackD if and only if each Type-1 part of the solution partially constitutes a total Type-1 solution.*

To find a Type-2 solution, we need to violate Rule 2 with small changes and we shall discuss the detail when giving our modified algorithms. However, to find Type-4 solutions is more complicated and requires a more thorough analysis.

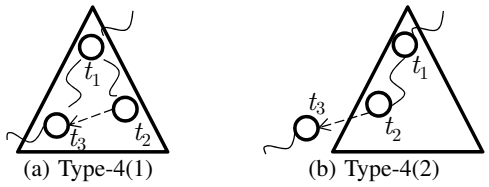
Function `getMissings` and `sweepParitalSolutionsTSD` are designed to discover descendant extensions in streams and pools respectively. Given a node t_q, function `getMissings` is able to find those child streams in which no nodes are TC descendants of t_q or those descendants have no extensions in the

Algorithm 3: TwigStackD [3]

```
1.  while ¬ end() do
2.      q_min := argmin_i{next(T_qi).start}; t_qmin := next(T_qmin)
3.      for each q_i in subtree(q) do
4.          while ¬empty(S_qi) ∧ top(S_qi.end) < t_qmin.start do
                pop(S_qi)
5.      missings := getMissings(q_min, t_qmin)
6.      if sweepPartialSolutionsTSD(q_min, missings) then
7.          if isroot(q_min)∨¬empty(S_parent(q_min)) then
8.              push(S_qmin, t_qmin), pointer to top(S_parent(q_min)))
9.              advance(T_qmin)
10.             if isLeaf(q_min) then
11.                 showSolutionsWithBlockings(S_qmin, 1, null)
12.                 pop(S_qmin)
13.         else advance(T_qmin)
14.     else advance(T_qmin)
15. mergeAllPathSolutions()
    Function: sweepPartialSolutionsTSD(q, missings)
16. for each q_i in child(q) do
17.     for each h in P_qi do
18.         if checkContainment(next(T_q), h) then
19.             candidateSet[q_i]:= candidateSet[q_i] ∪ {h}
20.             if q_i ∈ missings then missings := missings\{q_i}
21. if empty(missings) then
22.     for each q_i in children(q) do
23.         for each h in candidateSet(q_i) do expand(q, next(T_q), h)
24.     if isLeaf(q) then expand(q, next(T_q), null) // (*)
25.     return true
26. return false

    Function: getMissings(q, t_q)
27. missings := ∅
28. for each q_i in children(q) do
29.     pos := 1
30.     if checkInSync(q_i, pos, t_q) then
31.         allInSync := false
32.         while (pos<=size(T_qi))∧(T_qi[pos].start<t_q.end) do
33.             mi := getMissings(q_i, T_qi[pos])
34.             if ¬empty(mi) then pos++
35.             else
36.                 allInSync := true
37.                 break
38.         if ¬allInSync then
39.             missings := missings ∪ {q_i}
40.     else missings := missings ∪ {q_i}
41. return missings

    Function: checkInSync(childq, pos, tpar)
42. while (pos<=size(T_childq))∧(T_childq[pos].start<tpar.start) do
43.     pos++
44. if pos <= size(T_childq) ∧ T_childq[pos].start < tpar.end then
45.     return true
46. else return false
```



(a) Type-4(1) (b) Type-4(2)

Figure 5: Two types of path components of Type-4 twig solutions. The path t_1–t_2–t_3 is part of a Type-4 solution. The solid lines enclosed in each triangle denote reachability on the tree-cover and the dashed edges denote non-tree edges.

tree-cover. Note that the function only relies on the interval encoding to determine the ancestor-descendant relationship between nodes. All the operations in `getMissings` thus actually work on the tree-cover rather than the whole data graph. As a result, `getMissings` cannot find the Type-2, Type-3, and Type-4 path extensions with all nodes exclusively in streams, for which their reachability cannot be identified. The whole algorithm is unable to return the particular Type-4 total solutions having a component(s) as suggested in Figure 5(a). Recall the example shown in Figure 2. When the evaluation processes a_1 at the very beginning, a_1 clearly has two descendant extensions with nodes all in streams, namely (b_1, c_1) w.r.t. B and (d_1) w.r.t. D; but a_1 is not put in P_A, since the ancestor-descendant relationship between b_1 and c_1 cannot be determined by using the interval encoding in `getMissings`.

The other type of path components of a Type-4 solution is shown in Figure 5(b). TwigStackD is unable to identify such a descendant extension that consists of both stream node(s) and pool node(s), as `getMissings` only checks structural constraints among stream nodes, while `sweepPartialSolutionsTSD` limits itself in searching in pools. An example of this type is (b_1, c_1) for a_5 in Figure 2. a_5 is seen as missing an extension w.r.t. B and TwigStackD thus fails to find the Type-4 solution (a_5, b_1, c_1, d_1). As a solution, `getMissings` needs to look up pool nodes when recursively searching for descendant extensions of a stream node as done in `sweepPartialSolutionsTSD` except that none expanding operations currently need to be performed. In this case, TwigStackD also has the problem caused by the conflict between the two orders for putting nodes in stacks and pools as we have discussed for PathStackD. Intuitively, TwigStackD expands partial solutions in stacks and pools in opposite directions. In Figure 2, the partial solution (b_4, c_3) is constructed after the process of searching for and expanding descendant extensions of a_3 in pools is finished. The Type-4 solution (a_3, b_4, c_3, d_3) is consequently missed.

We next describe our first modified algorithm (Algorithm 4) for correcting TwigStackD. Clearly, a natural way to fix the problem regarding function `getMissings` is to (1) use Interval+SSPI rather than the interval encoding alone for determining structural relationship among descendants and (2) look up pool nodes besides stream nodes to search for descendant extensions. Given an incoming node t_q, `getMissings` traverses its TC descendants w.r.t each child of q and attempts to find their descendant extensions by invoking the new function `checkMissingDEsBySSPI`. For a stream node, function `checkMissingDEsBySSPI` first searches for descendant extensions by determining the reachability with pool nodes. If the node does not reach any, the function has to further recursively explore the structure of stream descendants. $isChecked$ and $hasMissingDEs$ are both global variables, respectively recording whether a node has been checked for its descendant structure (either when it is processed in the main procedure or when it is a stream node and checked in `checkMissingDEsBySSPI` for one ancestor) and whether it has descendant extensions w.r.t. every child query node. They are used to guarantee that no nodes are examined for more than once. Notice that `checkContainment` used in `sweepPartialSolutionsTSD` to answer reachability

Algorithm 4: Modified TwigStackD without Pre-Filtering

Modifications: (1) replace line 1 and 2–14 in Algorithm 3 with line 1 and line 3–16 below; (2) replace line 33–37 in Algorithm 3 to line 22–23 below, and add function `checkMissingDEsBySSPI` and function `reachBySSPI`; (3) insert line 19 below before line 22 in Algorithm 3; (4) replace function `expand` the same as in Algorithm 2 and change `showSolutionsWithBlocking` in the way similar to the modification for `showSolutions` in Algorithm 2 (omitted).

1. **while** $true$ **do**
2. ···
3. **if** empty($S_{q_{root}}$) $\wedge$ $eof(T_{q_{root}})$ **then** break
4. **if** $\neg isChecked[t_{q_{min}}]$ **then**
5. $missings$:= getMissings($t_{q_{min}}$)
6. **if** $\neg isChecked[t_{q_{min}}] \vee (isChecked[t_{q_{min}}] \wedge \neg hasMissingDEs[t_{q_{min}}])$ **then**
7. **if** sweepPartialSolutionsTSD($q_{min}, missings$) **then**
8. **if** $\neg isChecked[t_{q_{min}}]$ **then**
9. $hasMissingDEs[t_{q_{min}}] := false$
10. advance($T_{q_{min}}$)
11. **if** isLeaf(q_{min}) **then**
12. showSolutionsWithBlockings($S_{q_{min}}$, 1, null)
13. pop($S_{q_{min}}$)
14. **else** $hasMissingDEs[t_{q_{min}}] := true$
15. $isChecked[t_{q_{min}}] := true$
16. **if** $hasMissingDEs[t_{q_{min}}]$ **then** advance($T_{q_{min}}$)
17. mergeAllPathSolutions()

Function: sweepPartialSolutionsTSD($q, missings$)

18. ···
19. push$\big(S_q$, next(T_q), pointer to top($S_{\text{parent}(q)}$)$\big)$
20. ···

Function: getMissings(t_q)

21. ···
22. **if** checkMissingDEsBySSPI($T_{q_i}[pos], t_q$) **then** pos++
23. **else** $allInSync := true$; break
24. ···

Function: **reachBySSPI**(v_1, v_2)

25. **if** $\neg visited[v_2]$ **then**
26. $visited[v_2] := true$
27. **if** $v_1.start \leq v_2.start \wedge v_1.end \geq v_2.end$ **then return** $true$
28. **else if** $v_1.end < v_2.start$ **then return** $false$
29. **for** pos from 1 to size($PL[v_2]$) **do**
30. **if** reachBySSPI($v_1, PL[v_2][pos]$) **then return** $true$
31. **return** $false$

Function: checkMissingDEsBySSPI(t_q, r)

32. **if** $isChecked[t_q]$ **then**
33. **return** $hasMissingDEs[t_q]$
34. $isChecked[t_q] := true$
35. **for** q_i in children(q) **do**
36. $find := false$
37. **for each** h in P_{q_i} **do**
38. **if** reachBySSPI(t_q, h) **then** $find := true$; break
39. **if** $\neg find$ **then**
40. pos = pointer(T_{q_i}) + 1
41. **while** pos <= size(T_{q_i}) $\wedge$ $T_{q_i}[pos].start < r.start$ **do**
42. pos++
43. **while** (pos <= size(T_{q_i}))$\wedge$($T_{q_i}[pos].end < t_q.end$) $\wedge$ $\neg find$ **do**
44. **if** reachBySSPI($t_q, T_{q_i}[pos]$) $\wedge$ $\neg$checkMissingDEsBySSPI($q_i, T_{q_i}[pos]$)) **then**
45. $find := true$
46. **else** pos++
47. **if** $\neg find$ **then**
48. $hasMissingDEs[t_q] := true$
49. **return** $true$
50. $hasMissingDEs[t_q] := false$
51. **return** $false$

queries cannot be used for `checkMissingDEsBySSPI`. This is because the queries are then on two stream nodes that may be later put in pools where they are processed in `sweepPartialSolutionsTSD`. The add-and-remove operations involved in `checkContainment` probably remove predecessors that should be used as an intermediate node for answering reachability queries in `sweepPartialSolutionsTSD`. We thus give a new function `reachBySSPI` as the reachability query processing algorithm without add-and-remove operations.

The way to resolve the conflict between stack encoding and pool encoding is similar to that in PathStackD: (1) function `expand` is properly modified; (2) before PathStackD invokes this function, the current node being processed should be pushed into the stack. Additionally, we need to relax Rule 2 to Rule 4 so that Type-1 parts of possible Type-4 or Type-2 solutions are allowed to be constructed in stacks even though they may not constitute any Type-1 total solutions. Finally, the termination condition of the first stage should be changed as in the modification for PathStackD.

Rule 4. *A node t_q is pushed into S_q if and only if it has at least one descendant extension w.r.t. each child of q.*

Consider the example of Figure 2. (1) With the modifications, a_1 will be pushed into S_A since the reachability between two TC descendants b_1 and c_1 will be checked in `checkMissingDEsBySSPI` and answered by `reachBySSPI`. The path solution (a_1, d_1) then can be derived from stacks. When b_1 sweeps P_C and identifies c_1 as a descendant, the partial solution (c_1) will be expanded to (a_1, b_1, c_1) by replicating (a_1, b_1) constructed in stacks to pools. In the end, the two path solutions will be merged to (a_1, b_1, c_1, d_1). (2) When invoking the modified `getMissings` on a_5 and searching for descendant extensions of its TC descendant b_1, TwigStackD will look up c_1 in P_C and correctly find (c_1) as a extension of b_1, which in turn can be expanded as a extension of a_5 w.r.t. B. (3) Although (b_2, c_2) is not in a Type-1 solution, it will be encoded in stacks due to the new rule and moved to the pools when c_2 is processed so that it is able to be expanded with a_2.

Time Complexity. Compared with the time complexity of the modified PathStackD, the additional major time cost in the modified TwigStackD includes (1) the cost of checking descendant extensions for stream nodes, i.e. the cost of function `getMissings`, (2) the cost of enumerating the path solutions, (3) the cost of merge-join operations on individual path solutions. We first analyze the time cost of the modified `getMissings`. Function `reachBySSPI` is invoked at most $O(|T|^2_{max}|Q|)$ times. Since each call of `reachBySSPI` takes $O(|E_G|)$, the total time cost of performing `getMissings` during an evaluation is $O(|E_G||T|^2_{max}|Q|)$. [3] does not take into account cost(2) and (3). The cost(2) is linear in the size of path solutions. Given that the modified TwigStackD complies with Rule 4, all path solutions are merge-joinable and as a result, the cost(3) can be linear time in the sum of the size of path solutions and the size of final twig solutions. Therefore, in addition to other costs similar in PathStackD, the total time complexity of TwigStackD is $O(|E_G||T|^2_{max}|Q| + |V_G||E_G||T|_{total} + |T|^{|Q|}_{max})$.

Space Complexity. [3] overlooks (1) the space cost for constructing the pool encoding, and (2) the space cost for storing the inter-

mediate path solutions in order to perform merge-join operations on them. The space cost of pools is $O(|T|^2_{max}|Q|)$ with the same analysis in computing the cost for PathStackD. Since the size of path solutions is bounded by $|T|^2_{max}|Q|$, the total space complexity of TwigStackD is $O(|T|^2_{max}|Q|+\min\{|T|_{total}, path_{max}\})$.

Clearly, TwigStackD is not quite efficient. In particular, if the size of input stream nodes is large, it is prohibitively costly to check for descendant extensions among stream nodes even when the size of intermediate solutions and final answer are pretty small and manageable. Intuitively, consider evaluating a twig pattern query on a single root DAG. Assume the root has the same label with the root query node. It is the first stream node to be processed. As function `getMissings` is to find whether there are descendant extensions w.r.t. each child of the root query node among stream nodes, in this example it is essentially to determine whether there is such a solution to the twig on the entire DAG that it contains the root. In the worst case, `getMissings` performs exactly the same for finding a solution to a twig, which is just what TwigStackD is designed for.

Alternatively, we can extend TwigStackD by just incorporating a process that can exactly identify those stream nodes guaranteed to appear in a final solution, with which the input stream nodes having required descendant extensions can be selected and consequently `getMissings` and `sweepPartialSolutionsTSD` can be abandoned (the expanding operations in them should be retained and put in the main procedure). [3] proposes such a pre-filtering process that is capable of selecting the desirable nodes and can be employed in TwigStackD, which takes $O(|E_G|+|V_G|)$ time. With the pre-filtering process, the time complexity of TwigStackD is reduced to $O(|V_G||E_G||T|_{total} + |T|^{|Q|}_{max})$, the same as PathStackD, while the space complexity does not change.

6. AN UNDERLYING ASSUMPTION

An assumption is made but not explicitly specified in [3] for PathStackD and TwigStackD. It is assumed that the tree-cover used in the algorithms is generated by a depth-first graph traversal. The Lemma 3.2 in [3], as a key basis leading to the major results of the correctness of PathStackD and TwigStackD including Theorem 3.1 and Theorem 4.1 in [3], states that, given two nodes v_1 and v_2, if v_1 is reachable from v_2, either v_1 is a TC descendant of v_2 or the end value of v_1 is smaller the $start$ value of v_2. Intuitively, it states that all edges are directed from the right to left. The lemma does not hold if the tree-cover cannot be generated by a depth-first traversal, since in a traversal of the tree-cover for creating the intervals, the end value is assigned to v_2 when v_2 and all the TC descendants are visited; but v_1 probably is not yet visited and hence has a $start$ value larger than v_2, thereby contradictory to the lemma.

Our modifications are also based on this underlying assumption, since the partial solutions encoded in pools are only expanded with nodes to the right of those consisting of the solutions. Figure 6 gives a tree-cover of the data graph of Figure 1 and the corresponding intervals of each node. The tree-cover as well as the intervals is not able to be generated by a DFS. Using the intervals, PathStackD has not put any descendant extensions of a_3 into pools when processing a_3; then PathStackD cannot find two solutions (a_3, b_2, c_1) and (a_3, b_2, c_2).

However, [3] claims in the end of Section 2.2 that the approach proposed in [1] could be adopted to pick an "optimum" tree-cover, which is not necessarily able to be obtained by a DFS. When presenting the time complexity of PathStackD, [3] also gives a wrong analysis by approximating the length of a linear recursive call stack of function `checkContainment` to be the diameter of the data graph based on the situation where the interval numbers of nodes are created by a traversal on an optimum tree-cover. Indeed, the approach in [1] can be used as a heuristic way to reduce the size of SSPI. Yet it cannot be used in PathStackD and TwigStackD. We have shown that PathStackD fails to find a part of solutions using an optimum tree-cover shown in Figure 6.

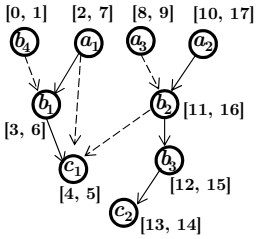


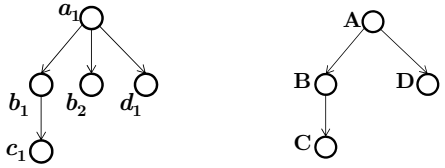
Figure 6: An optimum tree-cover of the graph in Figure 1

Algorithm 5: Modified Pre-Filtering

```
1.  for each node n in reverse topological order do
2.      for each child prev of n do
3.          if prev matches a query node ∧
            bitAND(nBitVector, preQBit)≠ 0 then
4.              reset := false
5.          else
6.              reset := true
7.          nBitVector := bitOR(nBitVector, preBitVector)
8.          if reset ∧ prev matches a query node ∧prev does not
            satisfy structural constraint then
9.              nBitVector := bitAND(nBitVector, ∼preQBit)
10.     if n matches a query node ∧
        bitAND(nBitVector, QBitVector) == QBitVector then
11.         n satisfies downward structural constraints
```

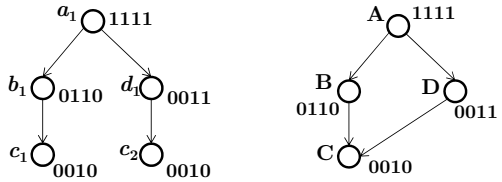
7. PRE-FILTERING PROCESS

[3] proposes a pre-filtering process to filter redundant nodes not appearing in solutions (Section 5 in [3]). Given a query pattern, two bit vectors with length equal to query size are assigned to each query node for respectively representing ancestors and descendants specified in the pattern. Specifically, each position Bit_q in a vector corresponds to a query node q; a bit vector for a query node q has 1 in position $Bit_{q'}$ if q' is the ancestor or descendant of q depending on the type of the vector. Such bit vectors can be also computed for data nodes, recording whether they has ancestors or descendants matching a particular query node. For example, in Figure 1, the bit vector of query node B for recording descendants is 011 (the three positions from left to right respectively represent A, B, and C). The same type of the bit vector for data node b_4 is also 011, indicating that b_4 satisfies the downward structural constraints for B. The general idea of the pre-filtering algorithm is to employ two graph traversals to get bit vectors for data nodes and filter those with bit vectors inconsistent with the vectors assigned to their corresponding query nodes. Algorithms 5 shows the process of a graph traversal for checking the downward structural constraints, where $QBit$ ($preQBit$) denotes a bit vector having 1 only in position corresponding to the query node that n (resp. $prev$) matches. The algorithm shown here is slightly different from the original algorithm in [3] in that we add a flag variable $reset$ to indicate whether a bit for the query node that $prev$ matches should be reset (line 3–6, line 8). When computing the bit vector for a node n, the original algorithm resets a bit Bit_q once n has a child $prev$ that has the label of q but does not satisfy structural constraints. However, the value of Bit_q may have been set before visiting $prev$ and now should



(a) Data graph (b) Twig pattern query

Figure 7: Example for illustrating the modification in the pre-filtering algorithm



(a) Data graph (b) DAG pattern query

Figure 8: Example for illustrating the pre-filtering process on a DAG pattern query

not be changed (e.g. the left sibling of $prev$ also has the same label and satisfies the structural constraints). Given a data graph and a twig query shown in Figure 7, consider the process of computing the bit vector of a_1 for checking downward structural constraints. Assume line 2 visits b_2 after b_1. Since b_2 has no descendants having the label of C, $QBit_C$ would be reset to 0 if literally following the original process, thereby a_1 incorrectly identified not satisfying structural constraints. In contrast, the refined process will not reset the bit since $reset$ that evaluates to false indicates that the value is inherited from b_1.

[3] claims that this pre-filtering process can identify nodes guaranteed to appear in the final results. The claim only holds for path and twig queries, but not for DAG queries. It is because each bit of a bit vector only records whether a ancestor or a descendant exist, but does not capture the specific structure among those ancestors and descendants. In a tree-structured pattern graph, there are no edges between descendants in different subtrees of a given node, so the downward structural constraints are satisfied by the node as long as all children also satisfy their constraints. Yet in a general graph-structured pattern, multiple nodes may share an identical child node, which means for a node, its different children may be required to have the same nodes. The fact that children respectively satisfy downward constraints does not lead to the conclusion that their parent node is also consistent with structural constraints. The reasoning line is similar when it comes to upward structural constraints. In Figure 8, b_1 and d_1 individually satisfy subtwig structural constraints, but a_1 does not, since b_1 and d_1 do not share a descendant with the label as C. In summary, pre-filtering is indeed useful for evaluating a path/twig pattern query, but it cannot filter all redundant nodes for a DAG pattern query. In addition, for a twig pattern query, because the process needs two graph traversals, it may be less efficient than directly performing `getMissings` and `sweepPartialSolutionsTSD`, when querying a large graph yet with a small size of stream nodes.

8. EXPERIMENTAL STUDY

In this section, we present an experimental study using both real and synthetic data to evaluate: (1) the effectiveness of the original PathStackD and TwigStackD; (2) the efficiency of the modified TwigStackD without and with the pre-filtering process; (3) the effectiveness of using the optimum tree-cover; and, (4) the effectiveness of the pre-filtering process on DAG pattern queries.

Table 1: Statistics of XMark datasets

Scaling factor	0.5	1	1.5	2	4
Dataset size (MB)	55	111	167	223	447
Nodes (Million)	0.64	1.29	1.94	2.52	5.17
Edges (Million)	0.77	1.54	2.32	3.09	6.20

Table 2: The total number of solutions returned by PathStackD-O and PathStackD-M

	$Path_1$	$Path_2$	$Path_3$	$Path_4$
PathStackD-O	19055	44422	188872	0
PathStackD-M	19055	44422	188872	89369

8.1 Experimental Setting

Datasets. We use large synthetic XMark data [5] and the real-life arXiv data in our experiments.

(1) We generated five XMark datasets with the scaling factors from 0.5 to 4. For each dataset, we constructed a node-labeled graph, where nodes correspond to XML elements, edges represent the internal parent-child links and ID/IDREF links, and labels are the tags of elements. The details are given in Table 1.

(2) We generated a graph from the HEPTh database[1], originally derived from the arXiv[2]. There are paper nodes and author nodes, each associated with multiple properties. We assigned a label to each author node according to the email domain, and a label to each paper node based on its area and journal it is published in. The edges of the graph represent author or citation relationships. The graph has 9562 nodes, 28120 edges, and 1132 distinct labels.

Queries. We used the query set shown in Figure 9 for XMark data, where a dashed edge indicates that the path between the two elements with the incident tags contains at least a ID/IDREF link. We randomly classified **person** and **item** elements into ten groups and each group has a distinct label. For each type of query, we generated ten different queries by randomly choosing a label for each **person** and **item** node.

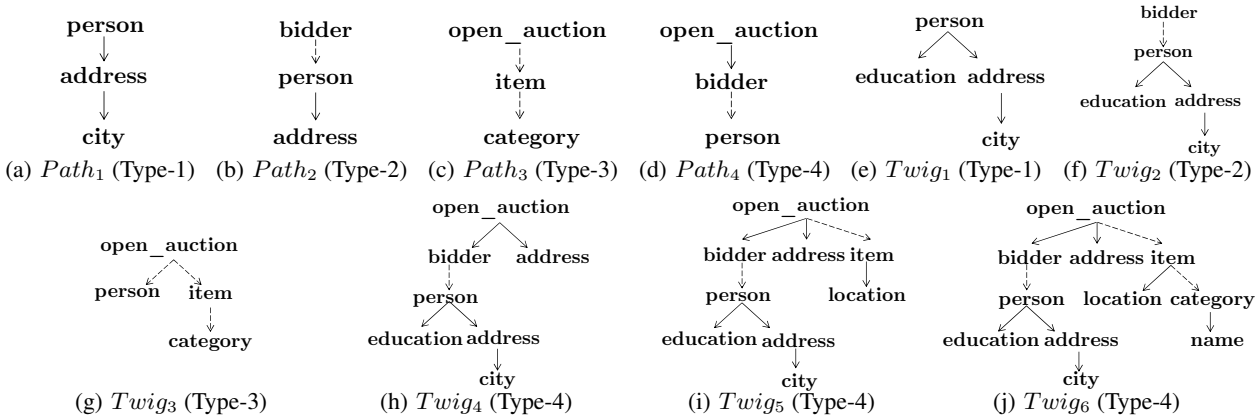
A query generator is also designed to produce twig and DAG queries for arXiv data. Each query node is associated with a label randomly chosen from the data graph. Two groups of queries are generated: one have a small size of results between 2 and 50, the other has a relatively large size of results between 200 and 1200. For each group, we generated 5 sets of queries with query size varying from 5 to 13. For each size scale, fifteen different queries are tested and the average is reported here.

Implementation. We have implemented the original PathStackD (denoted as PathStackD-O), the original TwigStackD (TwigStackD-O), the modified PathStackD (PathStackD-M), the modified TwigStackD without the pre-filtering process (TwigStackD-NF), and using the pre-filtering process (TwigStackD-F). In PathStackD-O and TwigStackD-O, the error with the termination time is corrected. All experiments are performed on a 2.4GHz Intel-Core-i3 CPU with 3.7 GB RAM running Ubuntu 11.04 (with gcc 4.5.2).

8.2 Experimental Results

Exp-1: Effectiveness of PathStackD. To evaluate the effectiveness of the original PathStackD, we used the queries of $Path_1$, $Path_2$, $Path_3$, and $Path_4$, whose solutions respectively belong

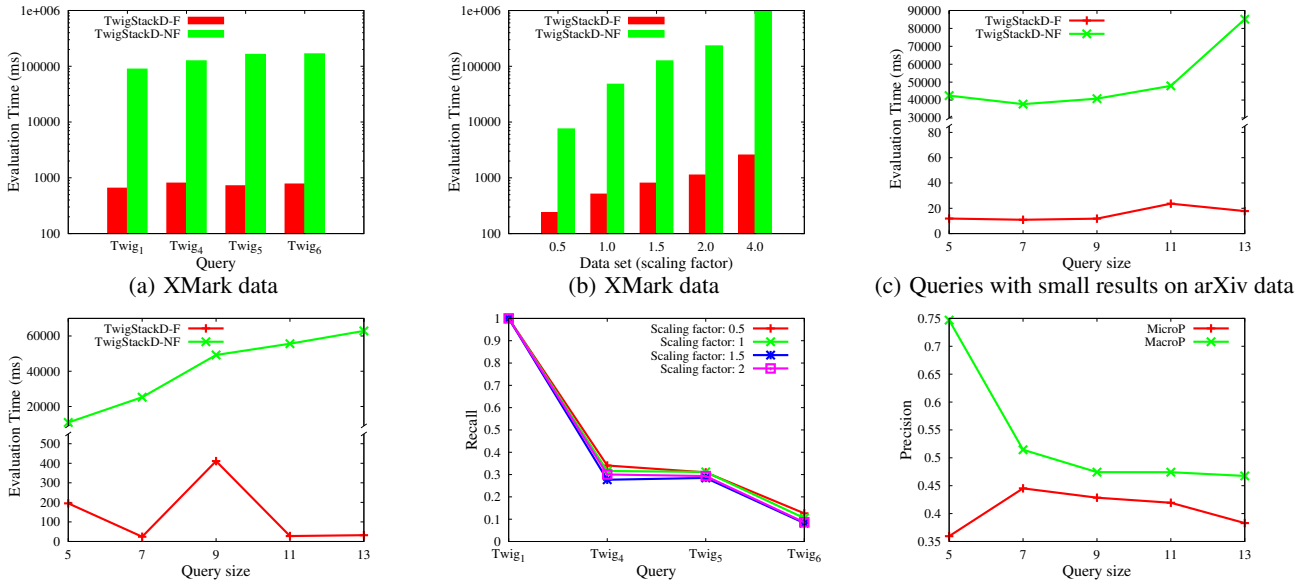
[1]http://kdl.cs.umass.edu/data/hepth/hepth-info.html
[2]http://arxiv.org



(a) $Path_1$ (Type-1) (b) $Path_2$ (Type-2) (c) $Path_3$ (Type-3) (d) $Path_4$ (Type-4) (e) $Twig_1$ (Type-1) (f) $Twig_2$ (Type-2)

(g) $Twig_3$ (Type-3) (h) $Twig_4$ (Type-4) (i) $Twig_5$ (Type-4) (j) $Twig_6$ (Type-4)

Figure 9: Queries for XMark data



(a) XMark data (b) XMark data (c) Queries with small results on arXiv data

(d) Queries with large results on arXiv data (e) Using the optimum tree-cover (f) Pre-filtering process for DAG queries

Figure 10: Experimental results

Table 3: The total number of solutions returned by TwgStackD-O and TwgiStackD-F

	$Twig_1$	$Twig_2$	$Twig_3$	$Twig_4$
TwgStackD-O	4794	0	18743	0
TwgiStackD-F	4794	11158	18743	11158

to Type-1, Type-2, Type-3 and Type-4. The total number of returned solutions to the ten generated queries for each query type on XMark data (scale 1.5) by PathStackD-O and PathStackD-M is presented in Table 2. The results show that PathStackD-O can find all solutions to $Path_1$, $Path_2$ and $Path_3$, but returns none to $Path_4$, since there are no Type-1 solutions to $Path_4$ and certainly no "lucky" cases of $Path_4$ where every Type-1 component of Type-4 solutions can be found in a Type-1 solution.

Exp-2: Effectiveness of TwigStackD. We evaluate the effectiveness of TwigStackD-O on the query sets of $Twig_1$, $Twig_2$, $Twig_3$ and $Twig_4$, whose solutions are respectively of Type-1, Type-2, Type-3 and Type-4. The results on XMark data (scale 1.5) are given in Table 3. TwigStackD-O fails to identify the solutions to $Twig_2$ and $Twig_4$. The reason why none of the Type-2 solutions and Type-4 solutions to $Twig_2$ and $Twig_4$ are not found is that the Type-1 components of those solutions are not in a total Type-1 solution. Indeed, $Twig_2$ and $Twig_4$ queries have no type-1 solutions. TwigStackD-O also failed to return any solutions to $Twig_4$, $Twig_5$ and $Twig_6$. The results show the inability of TwigStackD-O to find all Type-4 and Type-2 solutions.

Exp-3: Efficiency and scalability of TwigStackD. Since TwigStackD is a generalization of PathStackD, we focus on studying the efficiency and scalability of TwigStackD. In this set of experiments, we compare the performance of TwigStackD-F and TwigStackD-NF on both XMark data and arXiv data.

Figure 10(a) shows the evaluation time for processing $Twig_1$, $Twig_4$, $Twig_5$ and $Twig_6$ on XMark data with scaling factor 1.5. TwigStackD-F significantly outperforms TwigStackD-NF by more than two orders of magnitudes for all tested query sets. The processing time for both TwigStackD-F and TwigStackD-NF remains nearly the same with the increasing query size with reasons as following. (1) Due to the tree-like structure of XMark data, the size of the resulting SSPI is small and besides, most reachability queries can be efficiently answered by checking the intervals in constant time. As a result, The additional major cost for the operations of reachability query processing caused by the increased size of query

nodes is limited. (2) The size of solutions to $Twig_1$ (with an average of 479.4) is very small and the solutions to different query sets in this experiment are actually related. Intuitively, the evaluation process for $Twig_4$, $Twig_5$ and $Twig_6$ which contain $Twig_1$ involve little extra cost to derive solutions from the answer to $Twig_1$.

Figure 10(c) and 10(d) respectively depict the results of the group of queries with small results and the group with large results on arXiv data. In contrast to XMark data with a small density of nearly 1, the density of arXiv data is close to 3. The larger number of non-tree edges not only increase the size of SSPI, but also diminishes the importance of stack encoding and requires more costly operations of checking and expanding descendant extensions in pools. As a result, TwigStackD-NF is more sensitive to the query size on arXiv data. However, the elapsed time of TwigStackD-F does not change much, due to the effectiveness of the pre-filtering process, during which most nodes are pruned before constructing the stack and pool encoding. Yet TwigStackD-F fluctuates sharply in Figure 10(a). It has pool performance for particular queries with size of 5 and 9, for which the evaluation generates a relatively large size of intermediate partial results.

Figure 10(b) shows the evaluation time of the two algorithms for processing $Twgi_4$ on XMark data varying the data size. The results show that the larger the XMark data, the greater the gap between the performance of TwigStackD-F and TwigStackD-NF. While TwigStackD-F 30 times faster than TwigStackD-NF on XMark data with scaling factor 0.5, TwigStackD-F outperforms TwigStackD-NF by more than 300 times when the scale increases to 4.

The experiments reveal that TwigStackD-NF is inefficient and unscalable, but the pre-filtering process can considerably improve the performance. However, note that we do not take into account the I/O cost involved in the pre-filtering. Since the pre-filtering needs two graph traversals on the data graph and the graph databases in practical applications are often extremely large, TwigStackD-F is also supposed to be prohibitively costly in practice.

Exp-4: Effectiveness of the optimum tree-cover. In this set of experiment, we test the effectiveness of applying the optimum tree-cover to generate interval encoding and SSPI, using recall measure for quantitative comparison. Figure 10(e) shows the results of evaluating $Twig_1, Twig_4, Twig_5$, and $Twig_6$ on the four XMark data sets. The results are quite similar in different data sets. Because the paths between the elements matching the query nodes of $Twig_1$, such as the path from **person** elements to **education** elements, do not include ID/IDREF links in the original XML data, the TC ancestor-descendant relationship between them is maintained in any tree-covers including the optimum tree-cover. As a result, all solutions to $Twig_1$ are found. However, the recalls of other queries are less than 0.4. In particular, the more edges involving ID/IDREF links, the lower the recall. Therefore, the SSPI and intervals created on an optimum tree-cover cannot be used in PathStackD and TwigStackD unless the queries can be answered by PathStack and TwigStack, in which case the SSPI is actually not used.

Exp-5: Effectiveness of the pre-filtering process. This set of experiments uses the arXiv data to evaluate the effectiveness of the pre-filtering process in terms of the precision measure defined as follow:

$$\text{For each query } q_i, \text{precision}(q_i) = \frac{\#\text{resulting\_nodes}}{\#\text{selected\_nodes}}$$

Here #selected_nodes is the number of distinct nodes selected by the pre-filtering algorithm, and #resulting_nodes is the number of distinct nodes in the final query answers. We generated five DAG query sets with varying query sizes. For each query size scale, 15 different queries are tested and the micro-average and macro-average precision (MicroP and MacroP for short respectively) are reported in Figure 10(f). For most queries, the precisions are less than 0.5, which indicates more than a half of nodes returned by the algorithm are not in any query answer. The difference of MicroP and MacroP for 5-node DAG patterns is due to their different ways to present the measures. The precisions for a half of 5-node queries are nearly 1, but some are pretty small (less than 0.2). The MicroP tends to be affected by some queries for which the algorithm returns too many redundant nodes. The results verified that the pre-filtering step cannot exactly select the nodes guaranteed to appear in final results for DAG pattern queries.

9. CONCLUSIONS

Based on a classification of solutions proposed in this paper, we have shown that PathStackD cannot identify particular Type-4 path solutions and TwigStackD is unable to find particular Type-4 and Type-2 solutions. Both the missed Type-4 solutions and the missed Type-2 solutions are not rare in practice. A modified PathStackD and two modified algorithms for TwigStackd are proposed, which can guarantee soundness and completeness. We have shown that the time and space complexity of PathStackD and TwigStackD is quite high. In particular, in contrast to the claim in [3], the complexity is neither independent of the size of intermediate solutions and nor quadratic in the average size of the query variable bindings. We have also shown that the optimum tree-cover proposed in [1] cannot be used in PathStackD and TwigStackD, and the pre-filtering process proposed in [3] cannot exactly prune all redundant nodes not in the final answer to a DAG pattern query. We have experimentally verified the inability of PathStackD and Twigstack to find certain types of solutions, compared the efficiency of the two modified versions of TwigStackD, demonstrated the low effectiveness of the optimum tree-cover, and the low effectiveness of the pre-filtering process on DAG pattern queries.

Acknowledgements. We thank Gustavo Alonso, the anonymous reviewers and the authors of [3] for their helpful comments and suggestions. This work is supported by the National Science Foundation of China (61075074).

10. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 253–262, 1989.

[2] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 310–321, 2002.

[3] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on DAGs. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 493–504, 2005.

[4] J. Cheng, J. X. Yu, and P. S. Yu. Graph pattern matching: A join/semijoin approach. *IEEE Transactions on Knowledge and Data Engineering*, 23(7):1006 –1021, 2011.

[5] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: a benchmark for XML data management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 974–985, 2002.

[6] H. Wang, J. Li, J. Luo, and H. Gao. Hash-based subgraph query processing method for graph-structured XML documents. *Proceedings of the VLDB Endowment*, 1(1):478–489, 2008.

[7] H. Wang, J. Li, W. Wang, and X. Lin. Coding-based join algorithms for structural queries on graph-structured XML document. *World Wide Web*, 11(4):485–510, 2008.

[8] H. Wu, T. W. Ling, G. Dobbie, Z. Bao, and L. Xu. Reducing graph matching to tree matching for XML queries with ID references. In *Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part II*, pages 391–406, 2010.