Stack-based Algorithms for Pattern Matching on DAGs

Li Chen Amarnath Gupta M. Erdem Kurul

SDSC, University of California San Diego
9500 Gilman Drive, La Jolla, CA 92093-0505, USA
{lichen|gupta|erdem@sdsc.edu}

Abstract

Existing work for query processing over graph data models often relies on pre-computing the transitive closure or path indexes. In this paper, we propose a family of stack-based algorithms to handle path, twig, and dag pattern queries for directed acyclic graphs (DAGs) in particular. Our algorithms do not pre-compute the transitive closure nor path indexes for a given graph, however they achieve an optimal runtime complexity quadratic in the average size of the query variable bindings. We prove the soundness and completeness of our algorithms and present the experimental results.

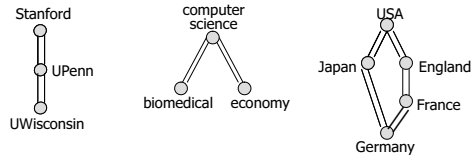
1 Introduction

1.1 Motivation

Graph-based database systems have been in existence for more than a decade. Recently, graph data models have been increasingly in demand by modern applications that utilize graph-structured data such as XML (if considering ID and IDREF), RDF and ontology data.

Surprisingly, graph data in many application domains can be represented as directed acyclic graphs (DAG). For example, the gene ontology data available at http://www.geneontology.org can be modeled as DAGs with nodes representing gene terms and edges denoting their *is-a* and *part-of* relationships. Now let's consider another example – a patent (or publication) citation network. Suppose each patent is represented by a node, and it has incoming edges from the patent nodes that it cites. Such a citation network is DAG-structured induced by the *cited-by* relationships assuming no cyclic references may occur. In this example, each patent node is uniquely identifiable by its number, and any property of user's interest (e.g., the patent's category, patentee's affiliation or address) may serve as a label type for users to specify filtering conditions on. In this sense, we are concerned with node-labeled (not edge-labeled) DAGs.

Suppose a citation link portal like *citeseer* supports accesses to such a DAG-structured data and provides an advanced search tool that enables to query the citation patterns, e.g., the citing tendencies across categories or disciplines, among institutions, or following certain geographical spreading patterns. For example, one may be interested in finding *"the patents of the computer science category which are directly or indirectly cited by patents of the biomedical category and by those of the economy category"*. This can be expressed using a twig pattern query as illustrated in Figure 1(b), where a double-edge denotes the *transitive* cited-by relationship. Figure 1(a) and (c) are examples of path and dag pattern queries respectively, inquiring the cross-institution and cross-country citation patterns. The path and twig queries have been extensively studied in the XML setting [3, 9, 2] where data are modeled as trees instead of graphs.



(a) cross-institutions (b) cross-disciplines (c) geographical dissemination

Figure 1: Path, Twig and Dag Pattern Queries

This work targets to provide efficient algorithms for processing such pattern queries on DAGs. An example DAG G, a twig query and its results are depicted in Figure 2. For illustration purposes, both the data and the query nodes are labeled with abbreviated letters, e.g., 'c' for "computer science", 'b' for "biomedical", etc. In addition, data nodes with the same label are

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

distinguished by the affixed numbers.

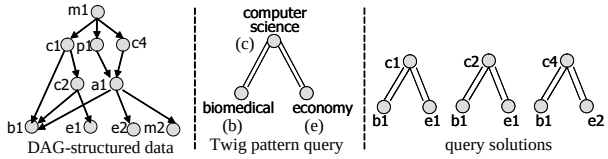


Figure 2: Example G, Twig Query, and Results

Here, we borrow and extend the XPath syntax as below to represent the pattern queries in Figure 1.

Step	::=	/ | // (*/ matches a parent-child relat.*)
NodeTest	::=	label
Path	::=	Step NodeTest | Step NodeTest Path
Twig	::=	Path | Path '('Twig, ..., Twig')'
Dag	::=	Twig (*allows duplicated node labels*)

In path and twig queries, each node has a unique label. Although appearing the same as a twig query, a dag query contains multiple occurrences of node labels to indicate where the paths meet. For example, the three queries in Figure 1 are expressed respectively as "//Stanford//UPenn//UWisconsin", "//computer_science(//biomedical, //economy)", and "//USA(//Japan//Germany, England//France//Germany)".

1.2 Related Work

In a recent survey [19], methods of pattern matching on graphs are categorized into *exact* and *inexact* matching. The exact matching requires a total mapping from query nodes to data nodes, i.e., *all* query nodes are exactly matched by their corresponding data nodes, and each parent-child "/" (resp. ancestor-descendant "//") query edge is mapped to an edge (resp. a path) in the data graph. The inexact matching allows "approximation" either via a partial mapping from query nodes to data nodes, or by transforming data nodes to establish a total mapping.

In this paper, we confine our work to the exact matching only. The exact graph matching problem is NP-complete in general [8]. However, efficient algorithms exist for some special cases. For example, polynomial algorithms in the size of the data graph exist [16, 18] when the data graph is acyclic. In a recent work [11], a near-quadratic algorithm in the data graph size is presented for tree and dag pattern queries on DAGs under the exact matching semantics.

As observed in [19], many graph searching systems utilize a pre-filtering step based on customized traditional indexes and estimation methods for reducing the search space before executing the graph matching algorithms. The most popular indexing methods in use are path-indexes with fixed or parameterized lengths [15, 17, 19].

For queries that involve ancestor-descendant (i.e., "//") edges, building path indexes for matching such "//" edges is equivalent to computing the path from every node to all of its descendants. This is known as the **transitive closure** of a graph. In the tradeoff between space and time, most existing graph matching approaches assume static data graphs and hence prefer to pre-compute the transitive closure or build variable-length path indexes to trade space for efficient pattern matching. The classic Floyd-Warshall algorithm [7] used for computing the transitive closure is $O(n^3)$ (n is the number of nodes in the graph) and the result storage cost is $O(n^2)$. In addition, a polynomial cost is incurred for maintaining the materialized transitive closure upon data updates.

In [5], the selectivity estimation problem is investigated for searching twiglets with only "/" edges in tree-structured data. The proposed selectivity estimation method is likely inefficient or imprecise when applied to the twiglets which contain also "//" edges.

The implementation of matching is classified in [19] as either by traditional (sub)graph-to-graph matching techniques, or by structural join approaches which decompose a query into a set of paths and then join the branching nodes of each derived path [2, 13, 20]. However, such structural join approaches often induce large intermediate results which may severely limit the query efficiency.

1.3 Our Approach

We are motivated to provide efficient graph matching algorithms with a minimal space cost. This work is inspired from the data streaming model exploited by the stack-based algorithms proposed in [3] for processing path and twig queries over the tree-structured XML data. In this model, a stream is associated with each query node (i.e., NodeTest) and it stores the corresponding satisfying data nodes (i.e., those match the label and are hence referred to as *query variable bindings*). In addition, a set of linked stacks are utilized to process each binding node only once, either discarding it or creating a pointer from it to the top node in its parent stack. These stack-based algorithms are linear in the total size of the query variable bindings and hence advantageous over those that read the entire input data. Also, they are superior to the structural join approaches for their effectiveness in dampening irrelevant intermediate results and capability of returning the final solutions in a non-blocking manner.

In this paper, we extend the original stack-based algorithms to handle path, twig and dag pattern queries on DAGs. The following key techniques are employed.

1. The DAG-structured data is stored in the form of a node table with interval encodings on a tree cover (i.e., spanning tree) of the graph and a customized predecessor index.
2. While still utilizing the data streaming model and linked stacks, the new algorithms employ an additional *partial solution pool* structure to selectively hold the stack-popped nodes. These nodes are then checked with each new read-in stream data to find possible transitive connections.

3. The new algorithms exploit the temporal properties in the node processing to guarantee that the returned solutions are sound and complete.
4. With only a linear storage cost for the data graph, our approach achieves an optimal query performance by adopting a pre-filtering step for pruning nodes to be put in streams and heuristics for quickly finding transitive connections.

Paper Outline. For the rest of the paper, we first present our DAG representation in Section 2 and then the stack-based algorithms extended for pattern matchings on DAGs in Sections 3 and 4. A pre-filtering step is described in Section 5. We show the experimental results in Section 6, and conclude in Section 7.

2 Representing DAG

In this work, we aim to develop time-efficient pattern matching algorithms for DAGs yet based on a space-efficient data storage. We assume that the given data graph G consists of a single rooted DAG; disjoint components can be hooked together by creating a virtual root. Formally, G is represented as $G = (V, \prec_d)$, where V denotes all the nodes and $\prec_d$ is a patrial order (aka *transitive reduction*) relation between node pairs. Intuitively, each directed edge $e =< a, b >$ in G implies $b \prec_d a$, assuming a is the parent and b is the child.

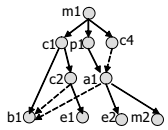
Several encoding schemes have been proposed for trees and DAGs, including bit-vector encoding [10, 4, 24], prefix encoding [12] and interval encoding [23, 1, 14]. Among them, the interval encoding scheme has recently attracted a lot of attention in XML research due to its overall performance in finding leaves, ancestors, descendants and nearest common ancestors, etc. [6]. In this scheme, each tree node is assigned with an integer pair $[start, end]$ (*level* may also be included) according to the visited orders in a depth-first traversal. In a tree, the ancestor-descendant (aka *transitive closure* $\prec$) relation between node pairs can be checked by their overlapping intervals [23]. For example, a node x is an ancestor of a node y, denoted $y \prec x$, if and only if $y.start > x.start$ and $y.end < x.end$.

Different from the tree case, a node y in a DAG may have more than one parent. Corresponding to such a multiple inheritance hierarchy of a DAG, [1] proposes to associate with a node multiple intervals that encapsulate reachability information for its descendant nodes. This multiple interval encoding method induces a $O(n^2)$ storage cost, which is the same as for directly storing the transitive closure relation. Although the space cost can be reduced by checking and discarding subsumed intervals, the worst-case storage required for the compressed closure is still $O(n^2)$ [1].

There is a line of research on further optimizing and compressing the storage costs of various encodings. However, we explore improvement opportunities from the query processing itself while employing space-efficient yet simple, adaptive storage structures.

In this spirit, we propose to represent a DAG G using a combination of interval encodings on a spanning tree T of G, called **tree-cover** in [1], and a customized predecessor index for nodes that are reachable from other nodes via the **remaining edges** E_R. E_R consists of edges in G excluding those covered by T. Namely, $E_R = E - E_T$ assuming E and E_T denote the edges in G and T respectively.

In Figure 3 (a), we illustrate a tree-cover T of the example DAG G in Figure 2 obtained via a depth-first traversal of G. The tree-cover edges are denoted by solid edges, while the remaining graph edges are depicted by dashed edges. By applying the interval encoding scheme [23] on this T, we derive the encoding for each node in G as show in Figure 3 (b).



(a)
⟶ for $e \in E_T$,
⟶ for $e \in E_R$

nid	*label*	*encoding*
m1	m	[1, 20]
c1	c	[2, 9]
b1	b	[3, 4]
c2	c	[5, 8]
e1	e	[6, 7]
p1	p	[10, 17]
a1	a	[11, 16]
e2	e	[12, 13]
m2	m	[14, 15]
c4	c	[18, 19]

(b)interval encoding on T

nid	*preds*
b1	{c2,a1}
a1	{c4}
e2	{a1}
m2	{a1}

(c) SSPI index

Figure 3: Illustration of DAG Representation

In a DAG G, any path p can be represented by $p := (e_t|e_r)^+ (e_t \in E_T, e_r \in E_R)$, i.e., p is formed by a mix of tree-cover edges and remaining edges. Suppose a path p_{mix} contains at least one remaining edge (i.e., the e_r type), then $p_{mix} := (e_t|e_r)^* e_{ri}\, e_t^*$ is modified from $p := (e_t|e_r)^+$, where e_{ri} is the last remaining edge (may also be the only one) followed by none or more tree-cover edges. The complementary type of such a p_{mix} path is a path that consists of purely tree-cover edges, denoted as $p_{pure} := e_t^+$. It is obvious that if a node y in G can be reached from a node x via a p_{pure} path, then $y \prec x$ and this reachability information can be revealed from their intervals, namely, $y.start > x.start \ \wedge \ y.end < x.end$. However, if x and y are connected solely by p_{mix} paths, then just the node intervals do not suffice to infer the full transitive closure $\prec$ of G.

2.1 Surrogate&Surplus Predecessor Index

In this section, we propose a new index that holds information complementary to the node intervals so that the full transitive closure $\prec$ of G can be derived. As explained earlier, the edge set E of G encapsulates a minimal relation $\prec_d$ whose transitive closure is $\prec$. Since $E = E_R + E_T$, the part of $\prec$ derivable from the subset of $\prec_d$ preserved in E_T is readily discoverable from the node intervals. This part of $\prec$ corresponds to all the p_{pure} paths.

Our goal is hence to utilize a space-economic index to store information that is enough for the rest of $\prec$ to be inferred. We refer to this remaining part of $\prec$ as $\prec_{rem}$, which corresponds to all the p_{mix} paths.

Storing all the node pairs connected by p_{mix} paths is equivalent to materialize $\prec_{rem}$, while storing just the node pairs connected by the remaining edges is not enough for deriving those p_{mix} paths that consists of both remaining and tree-cover edges.

Definition 2.1 *Suppose x and y are respectively the starting and ending nodes of a p_{mix} path in G, i.e., $y \prec_{rem} x$. Since p_{mix}:=$(e_t|e_r)^* e_{ri}\, e_t^*$, there must be a node w along the path such that $w = child(e_{ri})$.* [1] *We call w a **surrogate predecessor** of y if $w \neq y$. Otherwise, namely if e_{ri} is incident on y itself, we call $parent(e_{ri})$ an **immediate surplus predecessor** of y. We build a **surrogate**&**surplus predecessor index (SSPI)** to hold for each such y a sorted list of predecessors of both types, denoted **PL(y)**, in ascending order by their start interval values.*

Example 2.1 *In the SSPI shown in Figure 3 (c) for the example G in Figure 2, $PL(b1)$=$\{c2, a1\}$, both of which are immediate surplus predecessors of $b1$. While $a1$ is a surrogate predecessor for both $e2$ and $m2$.*

2.2 Properties of Our DAG Representation

Theorem 2.1 *For a DAG G represented using a node interval table by encoding on its tree-cover T and a SSPI index, the transitive closure $\prec$ can be losslessly derived from 1) the overlapping node intervals, together with 2) the predecessors stored in SSPI.* □

PROOF. It is obvious that the node intervals reveal the part of $\prec$ that corresponds to all the p_{pure} paths. We now prove that all the remaining $\prec_{rem}$ corresponding to the p_{mix} paths can also be inferred. Suppose that nodes x and y in G are connected by a p_{mix} path that has i edges of the e_r type, e.g., $p_{mix} = e_t^* \underline{e_{r1}} e_t^* \underline{e_{r2}} .. \underline{e_{ri}} e_t^*$. This $y \prec_{rem} x$ can be discovered by transitively looking up SSPI. That is, by first searching through $PL(y)$, the connecting node with e_{ri} (either the parent or the child) can be found. If this connecting node is $child(e_{ri})$, then we look up its PL for $w_i = parent(e_{ri})$. By recursively repeating the above step(s), we will finally find $w_1 = parent(e_{r1})$ whose interval is subsumed by that of x. □

Take Figure 3 as an example. $e2 \prec_{rem} c4$ is true in the example G, and it can be discovered by looking up $PL(e2)$ and then $PL(a1)$ in our SSPI structure.

Creation Time and Space Cost. Suppose a DAG G has n nodes and m edges ($m \geq n-1$). The creation time and space costs for our DAG representation are both in linear proportion to the size of G. Specifically, the cost break-down is as follows. First, it is obvious that both the time and space costs on the node interval table is $O(n)$. Second, for populating SSPI, we conduct a tree traversal of G. At each node w, we check its incoming edges that are of the e_r type and collect the corresponding immediate surplus predecessors into $PL(w)$. In addition, $PL(w)$ inherits all the predecessors from $PL(tree\_pred(w))$, where $tree\_pred(w)$ is w's parent node in the tree-cover (i.e., w is connected to it via the sole e_t type incoming edge). Therefore, the computation time for building SSPI is $O(m)$. Also, the total number of predecessors stored in SSPI in the worst case is bounded by m.

Compared to the adjacent list structure, SSPI does not necessarily preserve the exact transitive reduction relation $\prec_d$ of G. Instead, it stores a minimum subset of $\prec_d$ and $\prec$ whose total size is no greater than that of the adjacent lists so fewer computations may be involved in deriving all of $\prec$. Furthermore, heuristics can be applied to minimize the space of SSPI. For example, we can adopt the technique given in [1] to pick the "optimum" tree-cover of G which tends to span along longer paths and hence reduces the inherited predecessors in the PLs of SSPI.

3 Extending PathStack

In this section, we present our algorithm for path pattern matching on DAGs that is extended from the original stack-based PathStack algorithm [3] for trees. The challenge here is to extend the original algorithm to process the streaming-in nodes to derive query solutions that cannot be discovered by the stack operations. Our algorithm utilizes an additional structure collectively called *partial solution pools* besides the initial *streams* and the *stacks* in the matching process.

3.1 Partial Solution Pools

In the original PathStack algorithm, upon pushing a node into its corresponding stack, it pops from stacks the nodes that do not have overlapping intervals with the new node. That is, the stacks operate accordingly based on the tree-cover intervals. Once a node is popped from its stack, it is discarded immediately. By applying this algorithm directly to a DAG, e.g., the G in Figure 2, the second and third solutions at the rightmost part of Figure 2 will be lost. This is because that when $c2$ is pushed into its stack $S[c]$, $b1$ is already popped out of its $S[b]$ due to $b1.end < c2.start$. Hence the valid path $c2-b1$ cannot be discovered. Similarly, inducing the path $c4-b1$ (i.e., $b1 \prec c4$) would involve two remaining edges, which does not concern PathStack designed for pattern matching on trees.

In our DAG context, when $b1$ is popped from its stack due to the new incoming $c2$, we do not discard $b1$ right away but hold it in a structure to check for other potential solutions.

Definition 3.1 ***Partial Solution Pools*** *are a data structure we utilize to temporarily hold the stack-popped nodes to be grown from intermediate partial solutions to full solutions in a bottom-up fashion. Like stacks and streams, pools are created corresponding to*

[1] We use $parent(e)$ and $child(e)$ to represent the node where edge e is originated and the node e is incident on respectively.

each query variable for holding nodes that match the label. Two operations are associated with pools:

- *when pushing a node t_q into its stack S_{qi}, we* ***sweep*** *the child pool $Pool_{qj}$ (i.e., q_i=$parent(q_j)$ in the query pattern) to find with the aid of SSPI all such nodes t_{qj} that $t_{qj} \prec_{rem} t_q$.*
- *for each such t_{qj}, we build a parent pointer from it to t_q to* ***expand*** *the partial solutions headed by t_{qj} to those headed by t_q.*

The sweep operation makes sure that a node is put into its pool only if it finds descendants in the child pool. This hence guarantees a bottom-up expanding of partial solutions. When a node is put into the root pool corresponding to the query root variable, we output all the solutions headed by it as final solutions.

3.2 PathStackD

We now depict in Figure 4 the extended PathStack algorithm, called **PathStackD**, for handling *path* pattern queries on DAGs. A new sweepPartialSolutions procedure and a modified showSolutions procedure are the only modifications to the original PathStack[2].

3.2.1 Temporal Properties in Sweeping

In sweepPartialSolutions, an important temporal property between a new incoming node t_q (see line 01) and each node h in $Pool[child(q)]$ (line 02) is exploited.

Proposition 3.1 *At any point during the query processing, suppose t_q is the newly pushed node, y is any node remaining in the streams and x is any node in the pools. Then $x.start < t_q.start < y.start$.* □

The above property is determined by the data streaming model featured by the stack-based algorithms. Namely, function getMinSource makes sure that nodes in the streams are always retrieved and processed in their start value order. Nodes in the pools are those that have been processed and those in the streams have not, hence $x.start < t_q.start < y.start$.

Next we explain in detail how sweepPartialSolutions works in the aid of SSPI. As shown in the sweepPartialSolutions procedure in Figure 4, we iterate through each node h in the child pool of the new incoming node t_q, and call function checkContainment (line 05) to check whether $h \prec t_q$ by recursively looking up predecessors of h in SSPI.

Specifically, checkContainment iterates through each node $a \in PL(h)$ and concludes accordingly as in the following case analysis:

Case 1). a's interval is contained within t_q's (line 04). This means $a \prec t_q$ and hence $h \prec t_q$ is *true*.

Case 2). a's interval is not contained within but to the right of t_q's (line 06). Then $a \not\prec t_q$. Since predecessors in each PL entry of SSPI are sorted in ascending order by their start values (see Definition 2.1), we do not have to check the rest of predecessors in $PL(h)$ to conclude that $h \prec t_q$ is *false*.

Case 3). a's interval is not contained within but to the left of t_q's. In this case, we check if a itself is indexed in SSPI. If not (i.e., $PL(a) == null$ in line 08), then a can be removed from $PL[h]$. Otherwise (line 10), we recursively call checkContainment to check if any of a's one-level-upper predecessors has its intervals contained within t_q's. If all fails, we delete a from $PL[h]$ while shifting the unchecked predecessors in $PL[a]$ into $PL[h]$ (lines 11∼12).

In this sweeping process, the search space for checking $h \prec t_q$ in SSPI is tightly restrained in the following senses. First, t_q is checked with not any random

```
Algorithm PathStackD(q)
01 while ¬end(q)
02   qmin = getMinSource(q)
03   for qi in subtree(q) // clean stacks
04     while (¬empty(Sqi) ∧ top(Sqi).end < next(Tqmin).start)
05       pop(Sqi)
06   sweepPartialSolutions(qmin)
07   moveStreamToStack(Tqmin, Sqmin, pointer to top(Sparent(qmin)) )
08   if (isleaf(qmin))
09     showSolutions(Sqmin, 1, null)
10     pop(Sqmin)
```

```
Procedure sweepPartialSolutions(q)
01 tq = next(Tq)
02 for each h in Pool[child(q)] //check if h < tq via predecessors in SSPI
03   if (checkContainment(tq, h) == true)
04   expand(q, tq, h)
Function checkContainment(tq, h)
01 found = false;
02 while(¬empty(PL[h]) ∧ ¬found)
03   a = first(PL[h])      // get the first node a in PL[h]
04   if (a.start>tq.start ∧ a.end<tq.end)   // interval overlapping
05     return true
06   else if (a.start>tq.end)   // a is to the right of tq
07     return false
08   else if (PL[a] == null)   //a is to the left of tq & has no predecessor
09     remove a from PL[h]
10   else if ((found=checkContainment(tq, a)) == false)
11     add the remaining nodes in PL[a] into PL[h]
12     remove a from PL[h]
13 if (empty(PL[h]))
14   remove entry PL[h] from the predecessor lists
15 return found
Procedure expand(q, tq, h)
01 put tq into Pool[q]
02 h.ptr_to_parentPool = tq
03 if (isroot(q))      // if Pool[q] is the root pool
04   output the solutions headed by tq in the root pool
```

```
Procedure showSolutions(SN, SP, ChildSP)
01 index[SN] = SP // the position SP of stack S[SN] is interested
02 expand(SN, S[SN].index[SN], S[SN+1].ChildSP)
03 if (SN==1)
04   output(S[1].index[1], …, S[n].index[n])  // a root-to-leaf solution
05 else
06   for i =1 to S[SN].index[SN].ptr_to_stackParent
07     showSolutions(SN-1, i, SP)
```

Figure 4: Our PathStackD Algorithm

[2]Functions getMinSource, next(T_q), etc. are introduced in [3].

graph node but only nodes in its child pool. These checked nodes have smaller *start* values than t_q according to Proposition 3.1. Second, not all the stack-popped nodes but only those that find their descendants in the corresponding children pools are put in pools themselves. Third, according to the algorithm, even the child pool nodes are not exhaustively checked.

Also note that the size of SSPI is shrinking during the sweeping process. That is, a node is removed from a *PL* if it fails the checking of checkContainment and has no upper-level predecessors in SSPI. Furthermore, lines $13 \sim 14$ in checkContainment indicate that if $PL[h]$ is found to be empty when the recursive calls return (i.e., all the predecessors of h have been removed), then the $PL[h]$ entry is also removed. Such a shrinking implies a reduced search space for the subsequent containment tests. In sum, all these factors help to improve the efficiency of sweeping and hence of PathStackD. The detailed analysis is given next.

3.2.2 Analysis of PathStackD

We now introduce the following lemmas as the foundation for establishing the correctness of PathStackD.

Lemma 3.1 *During the checking for $h \prec t_q$ in checkContainment, suppose that a is removed from $PL[h]$ because $a.start < t_q.start \wedge PL[a] == null$ (case 3). Due to Proposition 3.1, the subsequent incoming nodes, e.g., t'_q, after a all have larger start values than a. Therefore, $a.start < t'_q.start$ always holds and the removal of a hence will not cause any solution loss.* □

The following lemma reveals the reason why the sweeping process guarantees to derive *all* the solutions complementary to those discovered by the original stack operations.

Lemma 3.2 *Given a pair of nodes x and y in G such that $y \prec x$, then either 1) $y.start > x.start \wedge y.end < x.end$ or 2) $y.end < x.start$. That is, y's interval is either contained within or to the left of x's.* □

Recall that Proposition 3.1 in [3] lists four possible cases of interval relationships between any pair of nodes. We now show that the other two cases that are not listed in Lemma 3.2 cannot co-exist with $y \prec x$. One case is $y.start < x.start \wedge y.end > x.end$. This implies $x \prec y$ which obviously conflicts with $y \prec x$ due to our DAG assumption. The other is $y.start > x.end$ (i.e., y's interval is to the right of x's). Assume $y.start > x.end$ is true, it means that y is encountered in the tree traversal of G only after the spanning tree rooted at x is fully traversed. This rules out the possibility that y is within the spanning tree rooted at x. Also, there can not be such a remaining edge that goes out from any node in the spanning tree of x to an ancestor of y because then y would be traversed to within the spanning tree of x and hence the conflict.

We hence have the following theorem.

Theorem 3.1 *Given a path query and a DAG, PathStackD correctly returns all the query answers.* □

We now analyze the time complexity of our PathStackD algorithm. As shown in sweepPartialSolutions in Figure 4, each node h in the child pool is swept through for a new incoming node t_q to check if $h \prec t_q$. During each checking, suppose the number of nodes being looked up in SSPI is c_i and the shrinking size is s_i. Then $s_i \approx c_i - d$ (if $c_i \geq d$, $s_i = 0$ otherwise), where d is the number of recursive calls made to look up predecessors in SSPI. This is induced from lines $02 \sim 14$ in checkContainment, where all the checked nodes, except those in the linear recursive call stack (see line 10), are removed. We may use the diameter of G (i.e., the longest length of all shortest paths among data nodes) to approximate d.

Since the total maximum shrinking size is the size of SSPI which is bounded by the number of graph edges m (see Section 2), we have $m \geq \Sigma_{i=1}^{|b|} s_i \approx \Sigma_{i=1}^{|b|}(c_i - d)$, where $|b|$ is the total size of query variable bindings in the input streams. Thus the total number of nodes being looked up in SSPI is $\Sigma_{i=1}^{|b|} c_i \leq m + |b|d$. In addition of the total number of pool nodes being checked, which is $\Sigma_{i=1}^{|b|} ch_i \leq |p_i|$ where $|p_i|$ is the size of a child pool at the time upon each new incoming node, the total cost is $m + |b|d + \frac{|b_i| \times |b|}{2}$, where $|b_i|$ is the average size of one input stream, and $|b|$ is the total of all stream sizes which can be approximated by $|q||b_i|$ ($|q|$ is the query pattern size). The factor of $1/2$ is because the size of a pool dynamically grows from zero to the full stream size. With the additional cost on outputing the final solutions whose size is bounded by $|b|$, we derive the following result:

Theorem 3.2 *Given a path query and a DAG, PathStackD has the worst-case[3] I/O and CPU time complexities of $O(|q||b_i|^2 + |q||b_i|d + m)$, i.e., $max(m, |q||b_i|(max(|b_i|, d))$). The worst-case runtime space complexity of PathStackD is $min(|b|, p_{max})$, where p_{max} is the maximum length of a path in the DAG.* □

The above theorem states that the worst-case time complexity of our PathStackD algorithm with respect to the input data is either in linear in the number of edges or quadratic in the average query variable binding size, whichever is larger. With respect to the input query, the time complexity is linear.

4 Twig Join Algorithm

Similarly, we generalize the original TwigStack algorithm to derive solutions for a twig pattern query for DAGs. Named TwigStackD, the extended algorithm does not use PathStackD as a sub-routine and hence

[3]The best case is when the input DAG is in fact a tree. PathStackD in this case executes like PathStack since no partial solution is build due to an empty SSPI.

avoids a suboptimal complexity caused by preserving non-final answers in the intermediate results.

4.1 TwigStackD

Algorithm TwigStackD is presented in Figure 5. Like TwigStack, it carries out operations in two stages. In the first stage (lines 01 ∼14), solutions to individual root-to-leaf paths are computed, then they are merge-joined in the second phase (line 15) to derive the final query solutions.

```
Algorithm TwigStackD(q)
01 while ¬end(q)
02   qmin = getMinSource(q)
03   missings = getMissings(qmin, next(Tqmin))
04   if (¬ isroot(qmin))
05     cleanStack(parent(qact), next(Tqmin))
06   if ((complete = sweepPartialSolutionsTSD(qmin,missings))==true)
07     cleanStack(qmin, next(Tqmin))
08     if (isroot(qmin) ∨ ¬empty(Sparent(qmin)))
09       moveStreamToStack(Tqmin, Sqmin, pointer to top(Sparent(qmin)))
10       if (isleaf(qmin))
11         showSolutionsWithBlocking(Sqmin, 1, null)
12         pop(Sqmin)
13     else advance(Tqmin)
14   else advance(Tqmin)
15 mergeAllPathSolutions()
Function getMissings(q, tq)
01 initialize missings as an empty set
02 for qi in children(q)
03   pos = 1
04   if (inSync=checkInSync(qi, pos, tq))==true)
05     allInSync = false
06     while(pos<=size(Tqi) ∧ Tqi[pos].start<tq.end ∧ ¬ allInSync)
07       mi = getMissings(qi, Tqi[pos])
08       if (¬empty(mi)) pos++
09       else allInSync = true
10     if (¬ allInSync)
11       missings.add(qi)
12   else missings.add(qi)
13 return missings
Function checkInSync(childq, pos, tparent)
01 while (pos<=size(Tchildq) ∧ Tchildq[pos].start<tparent.start)
02   pos++
03 if (pos<=size(Tchildq) ∧ Tchildq[pos].start<tparent.end)
04   return true
05 else return false
```

Figure 5: Our TwigStackD Algorithm

The key ideas of the first stage of TwigStackD are along the same line as those used in TwigStack. That is, a node is pushed into its stack only if it has all of the required descendant types. In TwigStack, the satisfiability of this property of a node t_q is ensured by checking: (i) if the interval of t_q contains that of at least one node t_{qi} in each of its children streams T_{qi} (we refer to the situation where a child is contained by its parent as "inSync"), and (ii) recursively check descendant streams to see if each t_{qi} satisfies the first condition. However, this property is relatively more difficult to be ensured for DAGs for the reasons below.

First, in the DAG setting, a descendant of t_q may exist in two forms: 1) in an input stream and not yet pushed into a stack, or 2) popped from the stack and put into a pool. TwigStack involves only the first case. *Second,* stream nodes that are "out-of-Sync" (as opposed to "inSync") cannot be simply advanced out of streams as in TwigStack since they may be used for expanding partial solutions. *Third,* the expanding of partial solutions is carried out only if t_q has all of the required descendant types, either located in partial solution pools or still in streams.

```
Function sweepPartialSolutionsTSD(q, missings)
01 for each qi in children(q)
02   for each h in Pool[qi]
03     if (checkContainment(next(Tq), h))==true)
04       candidateSet[qi].add(h)
05       if (qi ∈ missing)
06         missings.remove(qi)  //h <next(Tq) hence qi no longer missed
07 if (empty(missings))    // if all missings are complemented
08   for each qi in children(q)
09     for each h in candidateSet[qi]
10       expand(q, next(Tq), h)
11   return true
12 else return false
```

Figure 6: Sweeping Function for TwigStackD

Below we present a series of functions to resolve these difficulties. First, checkInSync is used to check whether the stream T_{childq} (i.e., T_{qi} since *childq* is passed with a child q_i of query variable q in getMissings) has any "inSync" node with respect to *tparent* starting from the position marker *pos* (initially set as *1* i.e., the top, at line 03 in getMissings). Function getMissings collects all the child streams which miss any of the required "inSync" children nodes or their descendant streams miss the required "inSync" descendants (lines 04∼12). If q_i is not in the *missings* returned by getMissings, then the node at the position marker *pos* in stream T_{qi} is an "inSync" descendant.

With the returned *missings*, TwigStackD calls sweepPartialSolutionsTSD (shown in Figure 6), which is modified from sweepPartialSolutions used by PathStackD, to check whether every missing child can be complemented by at least one node h in the corresponding partial solution pool. Only if this is true (line 07 in sweepPartialSolutionsTSD), partial solutions headed by h are expanded to be headed by the new node in $Pool[q]$ (lines 08∼10). Due to the space limitation, detailed explanations of these functions and of the modified showSolutionsWithBlocking are skipped.

TwigStackD guarantees that every solution to every individual root-to-leaf path is merge-joinable with at least one solution to each other root-to-leaf path. Hence the output individual solutions is no larger than the final solutions to the twig pattern query.

4.2 Analysis of TwigStackD

In this section, we discuss the correctness of TwigStackD and then analyze its complexity.

Definition 4.1 *Suppose for an arbitrary query variable node q in a twig pattern query, t_q is the top node*

in the stream T_q. If for each query variable node $q_i \in subtreeNodes(q)$, there is a data node t_{qi} in T_{qi} which is "inSync" with $t_{parent(qi)}$ and there is no such a node t'_{qi} that 1) t'_{qi} is also "inSync" with $t_{parent(qi)}$ and 2) $t'_{qi}.start < t_{qi}.start$, then we say that these t_{qi} nodes compose ***a minimal inSync descendant extension*** *of t_q.*

This concept can be seen an extension of a minimal descendant extension defined in Definition 4.1 in [3]. We establish the following lemma based on it.

Lemma 4.1 *Suppose that getMissings is invoked for an obtained q_{min}, then the following properties hold:*

- *the top node t_{qmin} of T_{qmin} has the minimal start value among all stream top nodes.*
- *upon returning missings, the nodes at the position marker pos of each stream $T_{qi}(q_i \in subtreeNodes(q_{min}) \wedge q_i \notin missings)$ form a minimal inSync descendant extension of t_{qmin}.*
- *if missings $\neq \emptyset$, sweepPartialSolutionTSD returns "true" if every missing child extension t_{qi} can be complemented by a node in $Pool[q_i]$, and "false" otherwise.* □

Based on this lemma, we can prove that if sweepPartialSolutionTSD returns true (line 06 in TwigStackD), then t_{qmin} is guaranteed to have a complete descendant extension composed of nodes in the descendant streams and in partial solution pools (detailed proof is omitted due to the space limitation). By pushing t_{qmin} into the stack, we can later build the parent pointers to it from its "inSync" descendants when they come in (line 09). This way, nodes that belong to a final solution will be preserved in the stacks to be output (lines 10∼11).

Theorem 4.1 *Given a twig pattern query and a DAG, TwigStackD correctly returns all the query answers.* □

Proof. In TwigStackD, we repeatedly get the stream T_{qmin} with its top node t_{qmin} satisfying the first property of Lemma 4.1. Assume that getMissings(q_{min}, t_{qmin})=M_{qmin}. For each query variable node $q_i \in M_{qmin}$, from lines 11∼12 in getMissings we know that the stream T_{qi} either has no "inSync" nodes to t_{qmin} or no such nodes that recursively have "inSync" descendants in the corresponding descendant streams. If $Pool[q_i]$ has no such a node connected to t_{qmin} after the sweeping (line 06 in TwigStackD), then t_{qmin} cannot possibly participate in any solution due to the lack of a descendant extension. We can thus advance it out (line 14) and continue with the next iteration. Otherwise, t_{qmin} has a full descendant extension composed of the minimal "inSync" descendant extension in the streams and partial solution pools. Line 08 ensures that t_{qmin} also has an ancestor extension, hence t_{qmin} definitely participates in at least one final solution so we push it into its stack (line 09). Finally, a root-to-leaf solution is obtained either when a node is pushed into its leaf stack (lines 10∼11) or put into the root partial solution pool (see expand in Figure 4). □

The key to analyzing the complexity of TwigStackD is to estimate the number of node checks (each count corresponds to an advance of the position marker *pos* by one node) in getMissings. Although each node may be repeatedly checked for different ancestors, it is guaranteed that the maximum repetition of checking the same node t_{qi} in a stream T_{qi} is bounded by the stream length. The rationale is given below.

Suppose getMissings(q_{min}, t_{qmin}) is invoked to recursively check nodes in the descendant streams. A node t_{qi} in each T_{qi} ($q_i \in descendant(q_{min})$) that sits above the position marker is checked. Such checking may be repeated for every node t_{qj} ($q_i \prec q_j \prec q_{min}$ in terms of the twig tree order) chosen by getMinSources prior to t_{qi}. Otherwise, t_{qi} will be advanced out of the stream earlier than t_{qj} and thus leave no chance for being checking again. Since there is at most one t_{qj} in each ancestor stream q_j that is a tree-cover ancestor of t_{qi}, the maximum number of checks for t_{qi} is $d_i - d_{min}$, where d_i and d_{min} are respectively the depths of q_i and q_{min} from the corresponding stream tops. Therefore, the maximum total node checks by getMissings is $\approx \Sigma_{i=1}^{|q|}(d_i - 1)|b_i| \approx \frac{h_q}{2}|q||b_i|$, where $|q|$ and h_q are the size and height of the twig pattern respectively. With the other costs similar as those for PathStackD, we derive the following complexity results for TwigStackD.

Theorem 4.2 *Given a twig query and a DAG, TwigStackD has the worst-case I/O and CPU time complexities of $O(|q||b_i|^2 + |q||b_i|(d + h_q) + m)$, i.e., $max(m,\ |q||b_i|(max(|b_i|, d, h_q))\ )$. The worst-case runtime space complexity of TwigStackD is the same as that of PathStackD, i.e., $min(|b|, p_{max})$.* □

This time complexity result is optimal compared to a latest work in [11] whose time complexity is $O(n|q||b_i|)$ for pattern matching on DAGs with a space cost of $O(n + m)$. The dominant factor in the time complexities of our algorithms is $|b_i|^2$ or m (whichever is bigger). Since $|b_i| < n$ and often $m < n|q||b_i|$ for a reasonably large query variable binding size, our algorithms are likely more efficient on average compared to the alternative algorithms in [11].

4.3 DagStackD

We also extend our TwigStackD algorithm with a filtering step for handling dag pattern queries. The basic idea is that for a "merge node" in the dag pattern (i.e., a query variable that has more than one incoming edges, for example, the node Germany in Figure 1), the *and* semantics must be enforced. Namely, we first

conduct the TwigStackD algorithm to find the solutions for the query requiring no *and* semantics yet. Then, for each merge node q_m in the dag query, we enforce the *and* semantics by checking whether each of its bindings t_{qm} has also participated in at least one solution to each other query path in the dag pattern that also includes q_m. If not, we filter it out similar to what is described in [11]. We adopt some heuristics in DagStackD to interleave such a filtering with the regular stack and pool operations to prevent the intermediate result nodes from overgrowing.

Due to the space limitation, we do not elaborate on the details of DagStackD here. The time complexity of DagStackD is that of TwigStackD with the addition of the complexity for joining the solution bindings at each query merge node among intersecting query paths. Its complexity is still in quadratic in the average size of the query variable bindings.

5 A Pre-filtering Step

The complexity analysis of our extended algorithms suggests that a pruning of the nodes in the streams can help to improve the runtime efficiency. In this section, we describe a novel pre-filtering technique that can prune the nodes to be put in the streams based on the structural pattern constraints of a query.

The abstract idea of pre-filtering includes two parts. First, we encode for each node in the query pattern the correspondingly *required* structural constraint. Second, we traverse the data graph and compute for each node a code value that indicates the aggregated *satisfied* constraints. If the *required* constraint is subsumed by the *satisfied*, then we put the data node is in the stream. Otherwise, we discard the node. Below we describe how the two types of encodings work.

5.1 Bit-vector Encoding of Query Pattern

For a given query, we assign a unique fixed-length bit-vector to each query variable node. The bit-vector length equals to the number of query nodes, and each bit position is designated to represent a particular query node. For example, we designate the i'th rightmost position to the query node q_i assuming that q_i's post-order is i during a depth-first traversal of the query pattern. For q_i, the bit at the position of itself (i.e., the i'th rightmost) is set as '1'. The other '1's are set at the positions corresponding to the post-orders of q_i's descendants. This way, the structural constraint on q_i in terms of the required descendant query nodes is reflected by its bit-vector. This is inspired from the bit-vector techniques used in [10, 4, 21].

5.2 Filtering Procedure

We employ two graph traversals to perform the node filtering. One traversal is for pruning the nodes that do not satisfy the *downwards* structural constraints while the other is for checking the *upwards* constraints. The graph traversals here are different from the one used for tree-cover encoding in section 2. The former ones visit each *edge* in the data graph exactly once while the latter visits each *node* exactly once. As illustrated in Figure 7, the structural constraints imposed by a twig query are expressed by the combination of two sets of bit-vectors. In the left top part, each query variable node is assigned with a bit vector value (denoted by $QBitVec$) that reflects all the required descendant query nodes. Note that the underlined '1' in a $QBitVec$ indicates that this bit position is the reserved one for this particular query node. In contrast, the set of $QBitVec$ values shown in the left bottom part reflect the structural constraints in terms of the corresponding ancestor query nodes required by the query nodes.

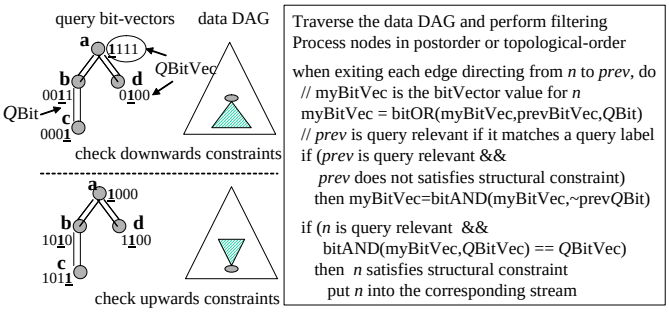


Figure 7: Illustration of Filtering Process

The node filtering process is depicted in the right part of Figure 7. Before the filtering starts, each data node is initiated with an all-zero bit-vector value. For checking the *downwards* structural constraints, we conduct a depth first traversal of the data graph to visit each edge exactly once and process the touched nodes in their *post-order*. Specifically, when leaving each edge e, we apply *bitOR* to the bit-vector value $myBitVec$ of $parent(e)$ (i.e., *prev* in Figure 7), the $prevBitVec$ of $child(e)$ (i.e., n), and $QBit$ whose only '1' bit is set at the position reserved for the query node corresponding to n. This ensures n to inherit the satisfied downwards constraint from the child node *prev*. The post-order processing guarantees that when computing the $myBitVec$ for a parent node n, the $prevBitVec$ of any child node is already finalized to be *bitOR*-ed. Then this $myBitVec$ is *bitAND*-ed with $QBitVec$ which encodes the required downwards constraint to check if the satisfied constraint of n subsumes what is required.

In the second graph traversal, nodes are processed in a *topological* order to be checked with the *upwards* structural constraints. Namely, x is processed prior to y for any directed edge from x to y. This ensures that the $myBitVec$ of any node is computed after those computations for all its ancestor nodes are done. The $myBitVec$ computation and checking with the corresponding $QBitVec$ here is the same as that in the first

traversal. Only if a node satisfies the constraints in both directions, we put it in its corresponding stream.

Although this pre-filtering technique helps to select the stream nodes guaranteed to appear in the final solutions, how exactly are the matched patterns (i.e., the final solutions with nodes linked to their particular ascendants or descendants) remains to be resolved by the stack-and-sweep process. The overhead of this pre-filtering step is two traversals of the data graph, i.e., $O(2m)$. We expect this overhead to be outweighed by the performance gains achieved due to the reduced stream sizes, especially when the data graph is large and the screening ratio is high.

6 Experimental Studies

In this section, we conduct experiments to evaluate our stack-based pattern matching algorithms for DAGs in comparison to the alternative linear-storage algorithms [11] (denoted by Nav in the experimental results). The basic idea of their algorithms in [11] is to first topologically sort the query nodes. Then for each query variable, an iteration over graph nodes is initiated to assign data bindings to it. For each such assigned node, the overall graph is searched for its child or descendant binding nodes. Both our algorithms and the compared algorithms do not require a time and space expensive path-index building nor transitive closure computing, as opposed to most algorithms surveyed in [19].

6.1 Experimental Setup

Our algorithms are implemented in Java 1.4 and use PSEPro from ObjectStore (http://www.objectstore.net) as the light-weighted storage engine for storing the DAG representation. The advantage of using PSEPro is that it provides us with not only the transparent persistency for the graph data but also a virtual memory mapping architecture (VMMA) which supports a direct mapping of the object hierarchy in memory onto the disk representation. This means that the accesses to the part of a data structure such as SSPI that may be stored on the secondary storage[4] are automatically translated by PSEPro to in-memory data accesses during execution. We conduct all the experiments on a 2.6Ghz Pentium IV PC with 1GB main memory and 2GB disk space allocated for the virtual memory.

We have devised a synthetic DAG generator with four tunable parameters: *diameter, fan-out, fan-in* and *number of distinct labels*. For example, the typical configurations for our synthetic data sets use *fan-out* and *fan-in* ranging from 2 to 20, *diameter* up to 20, and 10 to 50 distinct labels which are evenly distributed. For real-life data, we use a set of DAG-structured gene ontology data from the Gene Ontology Consortium and XML data generated from the XMark benchmark [22] with random additions of acyclic IDREFs. For all the conducted experiments, we have validated the soundness and completeness of our algorithms by comparing the output solutions with those produced by the alternative algorithms.

[4]The chance that SSPI is too large to fit in memory is much smaller than those space-expensive matching algorithms since SSPI's space cost is in linear (not quadratic) in the graph size.

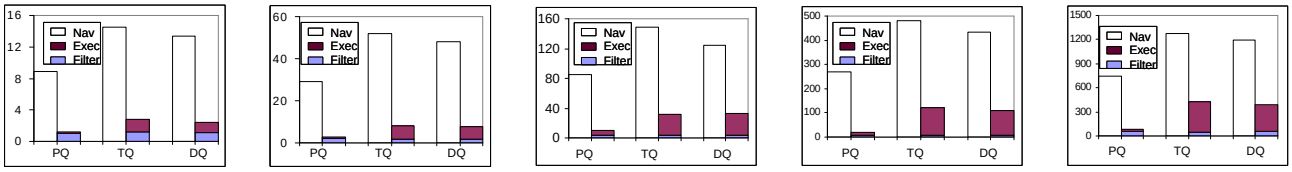
6.2 Experimental Evaluation

In Figure 8, we show the query performance in terms of response times (in seconds[5]) for different queries over increasing sizes of the synthetic DAG data sets. The different queries we use include a path query PQ=$//a//b//c//d$, a twig query TQ=$//a(//b(//d, //e), //c/\!/f)$, and a dag query DQ=$//a(//b/\!/d/\!/f, //e/\!/f)$. The DAG sizes increase from 25K nodes to 400K nodes, however with the ratio of the number of edges and that of the nodes fixed at 1.8, and the number of different labels set as 20. We can see that for all the DAG data sizes, the performance of our algorithms (shown in the second bar in each group) is always better than that of the alternative algorithms (denoted as Nav). The response time of each of our algorithms is composed of two parts, the pre-filtering time (denoted as $Filter$) and the query execution time (denoted as $Exec$). With each subsequent DAG doubling the size of its last one, the Nav algorithms show an increase by about four times in their response delays, while our algorithms show less pronounced effect of the DAG size. Since the pre-filtering cost is linear in the edge number of the DAG, its percentage weight in the overall query processing time reduces with the increase in DAG size.

Next we conduct experiments to show the query performance of both algorithms influenced by varying the densities of the DAG data sets. We start with a seed tree with 5K nodes and 10 different labels, then gradually increase the number of edges from $5K$ to about $70K$ by tuning the *fan-in* parameter of the DAG generator. Figure 9 shows that the response times to a twig query TQ=$//a(//b(//d, //e), //c/\!/f)$ increase with the growing ratio of the edges and the nodes for both algorithms. It also shows that our algorithms win over the Nav algorithms in all the cases, and that the improvements are more significant when the ratio is relatively large. We reason that this is because when the DAG is more dense, the search space for assigning data nodes as query variable bindings in the Nav algorithms is less likely restricted within local subtrees. This may impact their performance considerably. In contrast, our algorithms utilize the temporal properties and other techniques to help with space pruning and to avoid a severe performance decline.

Furthermore, we show the effects of path query length and twig query size on the query performance. The experimented path and twig queries are illustrated

[5]In all the experiments that show the time on the y-axis, the unit is *second*.



(a) 25K Nodes (b) 50K Nodes (c) 100K Nodes (d) 200K Nodes (e) 400K Nodes

Figure 8: Response Times of a Path, Twig and Dag Queries With Increasing DAG Sizes

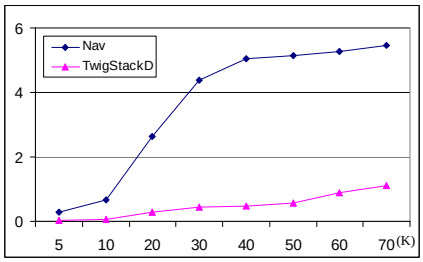


Figure 9: Effects of Graph Density

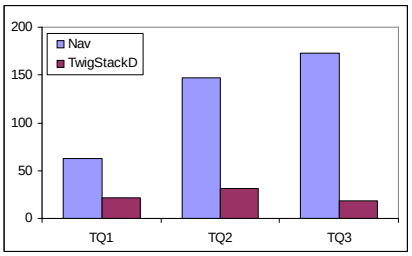


Figure 12: Effect of Twig Size

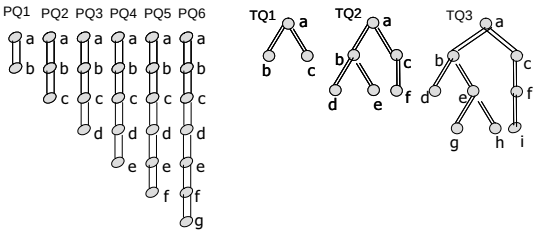


Figure 10: Queries used in Figures 11 and 12

in Figure 10. The test DAG data set has 100K nodes, 180K edges and 20 different labels. As shown in Figures 11 and 12, the processing time for the Nav algorithms increase with the growing of the path query length or the twig query size. Our stack-based algorithms however sometimes show performance improvements with more restricted query patterns (i.e., those with more query nodes). This is attributed to the pre-filtering step used for reducing the nodes to be put in the streams as well as to the decreased result size corresponding to the more restricted query patterns.

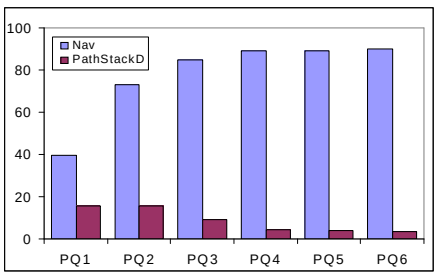


Figure 11: Effect of Path Length

To make sure that the performance gains brought by the pre-filtering step indeed outweigh its overhead, we compare the performance of our PathStackD algorithm (PSD for short) for a path query $PQ = //a//b//c//d$ with pre-filtering and without pre-filtering. Figure 13 shows that for a small DAG containing 1K up to 5K nodes (the edge/node ratio is 1.8 and the number of different labels is 20), the overhead of pre-filtering outweighs its benefits. As the DAG size increases to 25K or so, the performance of our PathStackD algorithm is better off to utilize pre-filtering.

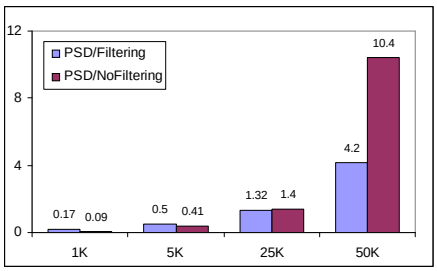


Figure 13: Effects of Prefiltering

In Figure 14, we report the SSPI building time for each DAG set used in the first experiment (see Figure 8) and the exact break-down of the processing time of TwigStackD (i.e., TSD) for the same twig query TQ. Also, we give out the number of nodes being touched (i.e., #Scan) by TwigStackD and the result sizes.

We have tested two sources of real data sets, one is the relatively small gene ontology data (17K nodes and 23K edges), and the other one is a 100MB XML document (about 1.4M nodes and 1.6M edges) produced by using the XMark [22] benchmark generator. For both real data sets, our stack-based algorithms out-

	BuildSSPI	TSDFilter	TSDExec	#Scan	#Result	NavAlgo
25K	1.02	1.26	1.58	0.58M	2.4K	14.5
50K	1.94	1.86	6.27	2.34M	5.6K	51.8
100K	4.38	3.84	28.35	10.9M	14K	148.7
200K	8.31	8.25	94.76	37.7M	24K	481.7
400K	19.06	21.11	375.90	121M	44K	1276

Figure 14: Scanned Nodes and Result Sizes

perform the Nav algorithms. For example, Figure 15 compares the performance of ours against Nav's for processing two particular queries over the real XML data set. Note that although the data size is large, the generated DAG is rather tree-like.

PQ=//site//person//age
TQ=//site(//item//description, //category//name, //person//age)

	*StackD	#Scan	#Result	NavAlgo
PQ	22.39	3.59M	12.8K	53.4
TQ	53.51	10.4M	27.8K	72.2

Figure 15: Experiments on Real XML Data

In sum, we have consistently observed the optimal experimental performance of our stack-based algorithms compared to the Nav algorithms. In particular, a restricted query pattern imposed over a dense graph is more likely favored by our algorithms.

7 Conclusions

In this paper, we have generalized the original stack-based algorithms to be applied to DAGs. With only a linear space cost for the data, our algorithms are quadratic in the average size of the query variable bindings in the worst-case time. A possible future work is to investigate whether the bit-vector encoding techniques can be extended and effectively used in our SSPI structure to enhance the node connectivity checking efficiency. We also plan on developing a general framework that allows for flexibly choosing from different options of DAG representations based on the trade-off between space and efficiency and on the structural properties of the particular input graph.

Acknowledgements. This work is supported by NIH Human Brain Project Award No. 5RO1DC03192 and NSF ITR Grant EIA-0205061 and NIH Grant P41 RR0088605 for NBCR.

8 References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD, Portland, Oregon*, pages 253–262, 1989.

[2] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE, Taipei, Taiwan*, pages 141–154, Feb. 2002.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD, San Jose, CA*, pages 310–321, June 2002.

[4] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *OOPSLA*, pages 271–287, 1993.

[5] Z. Y. Chen, H. V. Jagadish, F. Korn, and N. Koudas. Counting twig matches in a tree. In *ICDE, Heidelberg, Germany*, pages 595–604, 2001.

[6] V. Christophides, D. Plexousakis, M. Scholl, and S. Tourtounis. On labeling schemes for the semantic web. In *WWW*, pages 544–555, 2003.

[7] T. H. Cormen and et. al. *Introduction to Algorithms (ISBN 0-262-530-910)*. MIT Press, 1994.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, 1979.

[9] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing xml data for efficient structural joins. In *ICDE, Bangalore, India*, pages 253–264, March 2003.

[10] H. A. Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Trans. Program. Lang. Syst.*, 11(1):115–146, 1989.

[11] A. Kanza, W. Nutt, and Y. Sagiv. Querying incomplete information in semistructured data. *J. of Compu. and Sys. Sciences*, 64(3):655–693, 2002.

[12] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Symposium on Discrete Algorithms*, pages 954–963, 2002.

[13] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD, Madison, Wisconsin*, pages 133–144, 2002.

[14] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB, Roma, Itlay*, pages 315–326, Sept 2001.

[15] J. McHugh and J. Widom. Query optimization for XML. In *The VLDB Journal*, pages 315–326, 1999.

[16] A. O. Mendelzon and P. T. Wood. Finding Regular Simple Paths in Graph Databases. In *VLDB, Amsterdam, Netherlands*, pages 185–193, 1989.

[17] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[18] P. Buneman and M. F. Fernandez and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *The VLDB Journal*, 9(1):76–110, 2000.

[19] D. Shasha, J. T. Wang, and R. Giugno. Algorithmics and applications of tree and graph serarching. In *PODS, Maddison, WI*, pages 39–52, June 2002.

[20] Z. Vagena, M. M. Moro, and V. J. Tsotras. Twig query processing over graph-structured xml data. In *WEBDB, Paris, Frence*, pages 43–48, 2004.

[21] M. F. van Bommel and T. J. Beck. Incremental encoding of multiple inheritance hierarchies. In *CIKM, Kansas City, Missouri*, pages 507–513, 1999.

[22] Xmark. The xml-benchmark project. http://www.xml-benchmark.org, Apr. 2001.

[23] C. Zhang, J. Naughton, D. Dewitt, and Q. L. et. al. On supporting containment queries in relational database management systems. In *SIGMOD, Santa Barbara, California*, pages 425–436, 2001.

[24] Y. Zibin and J. Y. Gil. Efficient subtyping tests with pq-encoding. In *OOPSLA*, pages 96–107, 2001.