Efficient Algorithms for Pattern Matching on Directed Acyclic Graphs

Li Chen Amarnath Gupta M. Erdem Kurul
San Diego Supercomputer Center
9500 Gilman Drive, La Jolla, CA 92093-0505, USA
{lichen|gupta|erdem@sdsc.edu}

Abstract

Recently graph data models have become increasingly popular in many scientific fields. Efficient query processing over such data is critical. Existing works often rely on index structures that store pre-computed transitive relations to achieve efficient graph matching. In this paper, we present a family of stack-based algorithms to handle path and twig pattern queries for directed acyclic graphs (DAGs) in particular. With the worst-case space cost linearly bounded by the number of edges in the graph, our algorithms achieve a quadratic runtime complexity in the average size of the query variable bindings. This is optimal among the navigation-based graph matching algorithms.

1. Introduction

Recently, graph data models have been increasingly in demand by applications that utilize semi-structured data, ontology data, and RDF data. Surprisingly, graph data in many application domains can be represented as directed acyclic graphs (DAGs). We are hence motivated to develop fast searching algorithms for DAG-structured data, such as the anatomy ontology database illustrated in Figure 1. An example query may be *"find all the anatomies that are contained in brain and isa compartment"*. If we borrow a fragment of XPath expressions and extend it with "{}" to enclose edge label constraints, it can be expressed as *brain/{has*}/$a/{isa}/compartment* corresponding to the left path in Figure 1. If a constraint is added requiring that there must be an additional path from "brain" to neuron $\xrightarrow{isa}$ cell through a sequence of "has" edges, we can express the condition as *brain[{has*}/neuron/{isa}/cell]/{has*}/$a/{isa}/compartment* corresponding to the whole twig pattern in Figure 1 (edge labeled "*" indicates recursive transitivities).

The matching patterns addressed in this paper are restricted to exactly the same classes of path and twig patterns handled in [2]. However, pattern matching for DAGs is more complicated than for trees because multiple paths may exist between a pair of nodes in a DAG.

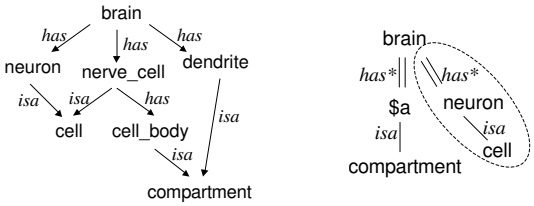


Figure 1. Example DAG and Pattern Queries

We extend the original stack-based algorithms proposed in [2] for handling pattern queries on XML trees to be applicable to DAGs by further exploiting useful properties in stack operations and result constructions. With the goal of efficiently checking the ancestor-descendant relationship that may be induced by multiple paths between a pair of nodes in a DAG, we utilize the following key techniques to prune the search space.

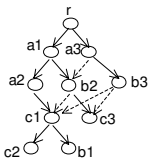
1. We apply the interval encoding scheme on the *tree cover* of the input DAG, and construct *predecessor lists* to index the remaining linking structure.
2. We propose the idea of using so-called *partial solution pools* to temporarily hold the processed nodes, which may be connected to subsequent nodes to grow into complete solutions.
3. We prune the search space for deriving all possible solutions by utilizing the temporal properties in expanding partial solutions.

2. Representing DAG

We propose to represent a DAG G=(V, E) using a combination of tree encoding and remaining graph edge indexing. Namely, we apply the interval encoding scheme [1] to a spanning tree T (called **tree-cover**, a term borrowed from [1]) of G and build an index for the edges in the **remaining graph** G'=(V', E'), where E'=$E-E_T$ (E_T is the edge set covered by T) and V' is the set of nodes connected by edges in E'. In the interval encoding scheme, each tree node

Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)
1084-4627/05 $20.00 © 2005 **IEEE**

is assigned with an integer pair *(start, end)* according to the visited orders in a depth-first traversal. This way, the hierarchical containment relationship between a node pair can be reflected by their overlapping intervals. Namely, query $a//b$ is translated to $a.start < b.start \wedge a.end > b.end$.



node_id	*label*	*encoding*
r	r	(1, 20)
a1	a	(2, 15)
a2	a	(3, 10)
c1	c	(4, 9)
c2	c	(5, 6)
b1	b	(7, 8)
b2	b	(11, 14)
c3	c	(12, 13)
a3	a	(16, 19)
b3	b	(17, 18)

node	*preds*
c2	{c1}
b1	{c1}
c1	{b2, b3}
c3	{b2, b3}
b2	{a3}

Figure 2. Tree-cover Encoding and PL of a DAG G

We also build an index structure called **predecessor lists** (PL) for nodes that can be reached via a path consisting of non-tree-cover edges. Specifically, for each such node, the PL indexes its "other" parents than the one revealed by the tree-cover T, if any, and also its closest ancestors that have "other" parents. An example DAG and its representation in terms of the encoding and the PL are illustrated in Figure 2.

3. Extending the Stack-based Algorithms

In the original stack-based algorithm, nodes in the streams are processed in their document order. Each node is processed only once, being either pushed into the stack to be linked to a parent node or discarded.

This approach however cannot be used to discover the structural containment induced by the cross links in the remaining graph of a DAG. In other words, we cannot discard a node n immediately when its parent stack is empty. There may exist a structural containment between a subsequent node and n due to cross links. To deal with this, we suggest to hold the processed nodes temporarily in a so-called *partial solution pool* data structure. For each subsequent node y, we "sweep" the corresponding child partial solution pool and conduct the reachability tests between y and the nodes in the pool. For each successful test between y and a pooled node u, we connect u to y. In this sense, we expand the partial solution referenced by u to the one by y. This way, partial solutions grow continuously in a bottom-up fashion. If a solution has grown to be headed with a root pool node, we output it as a final solution. The general idea is depicted in Figure 3. *Partial solution pools* are correspondingly labeled to indicate the types of the pooled nodes.

Although we essentially perform transitive closure computations in the sweeping process, the search space is pruned compared to an exhaustive search over the whole graph. The meaning is three-fold. First, we do not conduct containment checking for any two random nodes. Instead, the checking is only between a new node and the prior processed child pool nodes. Hence, at an early stage when a small portion of data bindings have been processed and put into their corresponding pools, the checking is confined to with them only. Second, not all nodes but only those which have full suffix partial solutions are pooled. Third, the predecessors (directed or recursively induced) of a child pool node are not exhaustively checked. We stop the checking when a predecessor has its interval to the right of that of the new node, because the ordered predecessors guarantee that no more successful containment cases can be derived. Furthermore, we apply heuristics which can shrink predecessor lists and further reduce the reachability computations.

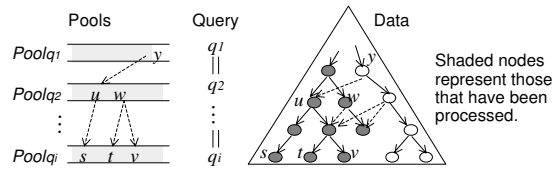


Figure 3. Illustration of Partial Solution Pool

In sum, our sweep procedure ensures that the containment cases left out by the regular stack operations will *all* be discovered between each new node and the prior processed in-pool nodes. In addition, our algorithms are advantageous among the navigation-based graph matching algorithms in terms of the time complexity.

Theorem 3.1 *Given a path or a twig query q and a DAG G, our extensions to the original stack-based algorithms can be applied to correctly return all answers for q on G.* □

Theorem 3.2 *Given a path query q and a DAG G, our PathStack extension has worst-case I/O and CPU time complexities of $O(|b_i|^2+m+|b|d+|r|)$[1]. For a twig query q', our TwigStack extension has worst-case I/O and CPU time complexities of $O(|b_i|^2+m+|b|(d+h_q)+|r|)$ [2].* □

References

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

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

1 $|b_i|$ is the average size of the data stream, m is the number of edges of G, $|b|$ is the total of all stream sizes, d is the graph diameter, and $|r|$ is the final solution size.

2 h_q is the height of q'.