Chapter 6

GRAPH REACHABILITY QUERIES: A SURVEY

Jeffrey Xu Yu
The Chinese University of Hong Kong, China
yu@se.cuhk.edu.hk

Jiefeng Cheng
The Chinese University of Hong Kong, China
jfcheng@se.cuhk.edu.hk

Abstract There are numerous applications that need to deal with a large graph, including bioinformatics, social science, link analysis, citation analysis, and collaborative networks. A fundamental query is to query whether a node is reachable from another node in a large graph, which is called a reachability query. In this survey, we discuss several existing approaches to process reachability queries. In addition, we will discuss how to answer reachability queries with the shortest distance, and graph pattern matching over a large graph.

Keywords: Graph, Reachability, Coding, Graph Pattern Matching.

1. Introduction

Graph structured data is enjoying an increasing popularity as web technology and archiving techniques advance. Numerous emerging applications need to work with graph-like data due to its expressive power to handle complex relationships among objects. Instances include navigation behavior analysis for web usage mining [3], web site analysis [22], and biological network analysis for life science [33]. In addition, *RDF* allows users to explicitly describe semantic resources in graphs [6]. Querying and analyzing graph structured data becomes important. As a major standard for representing data on the World-Wide-Web, *XML* provides facilities for users to view data as graphs with two

C.C. Aggarwal and H. Wang (eds.), *Managing and Mining Graph Data*,
Advances in Database Systems 40, DOI 10.1007/978-1-4419-6045-0_6,

different links, the parent-child links (document-internal links) and reference links (cross-document links), where the cross-document links are supported by value matching using ID/IDREF in *XML*. *XLink* (*XML* Linking Language) [19] and *XPointer* (*XML* Pointer Language) [20] provide more facilities for users to manage their complex data as graphs and integrate data effectively. The dominance of graphs in real-world applications demands new graph data management so that users can access graph data effectively and efficiently.

Graph reachability (or simply reachability) queries, to test whether there is a path from a node v to another node u in a large directed graph, have being studied [1, 24, 17, 28–30, 23, 13, 34, 32, 9, 14, 5, 26, 25, 10] and are deemed to be a very basic type of graph queries for many applications. Consider a semantic network that represents people as nodes in the graph and relationships among people as edges in the graph. There are needs to understand whether two people are related for security reasons [2]. On biological networks, where nodes are either molecules, or reactions, or physical interactions of living cells, and edges are interactions among them, there is an important question to "find all genes whose expressions are directly or indirectly influenced by a given molecule" [33]. All those questions can be mapped into reachability queries. The needs of such a reachability query can be also found in *XML* when two types of links (document-internal links and cross-document links) are treated the same. Recently, [8, 12, 35] studied graph matching problem on large graph data, where nodes in a match are connected by reachability relationships. Reachability queries are so common that fast processing is mandatory.

Reachability Queries: Let $G = (V, E)$ be a large directed graph that has n nodes and m edges. A *reachability queries* is denoted as $u \rightsquigarrow v$, where u and v are two nodes in G. Here, $u \rightsquigarrow v$ returns true if and only if there is a directed path in the directed graph G from u to v. In other words, let TC be the edge transitive closure of graph G, $u \rightsquigarrow v$ is true if and only if $(u, v) \in TC$. We call such a pair (u, v) a connection. Note: TC can be very large for a large and dense graph G. A reachability query over a directed graph G can be answered over a corresponding directed acyclic graph (DAG) of the graph G based on strongly connected components. Two nodes, u and v, are said to be in a strongly connected component, if and only if both $u \rightsquigarrow v$ and $v \rightsquigarrow u$ are true. And in a strongly connected component, for every two nodes, u and v, $u \rightsquigarrow v$ and $v \rightsquigarrow u$ are true. Given a directed graph $G(V, E)$, its strongly connected components, $C_1, C_2, \cdots$, can be efficiently identified in $O(n+m)$ time [18]. A DAG of the graph G, denoted G', can be constructed as follows. First, a strongly connected component C_i in G is replaced by a representative node v in G'. Second, all the edges between the nodes in the strongly connected component C_i are removed while all incoming edges and outgoing edges of C_i will be represented as incoming edges and outgoing edges of the representative node v in G'. A reachability query, $u \rightsquigarrow v$, over G can be processed over the

Table 6.1. The Time/Space Complexity of Different Approaches [25]

	Query Time	*Index Construction Time*	*Index size*
Transitive Closure [31]	$O(1)$	$O(nm)$	$O(n^2)$
Tree+SSPI [8]	$O(m-n)$	$O(n+m)$	$O(n+m)$
GRIPP [32]	$O(m-n)$	$O(n+m)$	$O(n+m)$
Dual-Labeling [34]	$O(1)$	$O(n+m+t^3)$	$O(n+t^2)$
Tree Cover [1]	$O(\log n)$	$O(nm)$	$O(n^2)$
Chain Cover [9]	$O(\log k)$	$O(n^2+kn\sqrt{k})$	$O(nk)$
Path-Tree Cover [26]	$O(\log^2 k')$	$O(mk')$ or $O(nm)$	$O(nk')$
2-Hop Cover [17]	$O(m^{1/2})$	$O(n^3 \cdot \|TC\|)$	$O(nm^{1/2})$
3-Hop Cover [25]	$O(\log n+k)$	$O(kn^2 \cdot \|Con(G)\|)$	$O(nk)$

DAG G' by checking whether the corresponding strongly connected component, where v resides, is reachable from the corresponding strongly connected components, where u resides. In the following, without otherwise specified, we assume G is a DAG.

There are two possible approaches to process a reachability query, $u \rightsquigarrow v$, in a graph G. It can be processed as to traverse from u to v using breadth- or depth-first search over the graph G on demand, when a reachability query is issued. It incurs high cost as $O(n+m)$ time. On the other hand, it can be processed as to check whether (u,v) exists in the edge transitive closure of the graph G, TC, by precomputing and maintaining the edge transitive closure TC on disk. It results in high storage consumption in $O(n^2)$. The two approaches are infeasible. The former requires too much time in querying and the latter requires too much space.

In the literature, many approaches have been proposed to reduce the space consumption, and at the same time answer reachability queries efficiently. Recall that by precomputing and maintaining the edge transitive closure TC of G, it can answer a reachability query in $O(1)$ time at the expense of $O(n^2)$ space. Here, the edge transitive closure TC servers as an index to be used to answer reachability queries. The existing approaches attempt to increase the query processing time marginally in the range of $O(1)$ and $O(n+m)$, where $O(1)$ is the query time using the edge transitive closure TC and $O(n+m)$ is the query time using breadth- or depth-first search, by constructing an index that can significantly reduce the space consumption. For example, some approaches construct an index based on a spanning tree of the graph G plus some additional information to maintain reachability information over the graph G, and some construct an index that compresses the edge transitive closure TC. On this direction, the time of spending on constructing an index becomes an important issue too.

Table 6.1 shows a summary on the time/space complexity of different approaches [25]. Given a graph $G(V,E)$. Let $n = |V|$ and $m = |E|$. Simon

proposes an algorithm to compute the edge transitive closure for a DAG, G, in $O(nm)$ time [31]. In other words, the time to construct an index based on the edge transitive closure of G is in $O(nm)$ time, and the index size is in $O(n^2)$ space, in the worst case. With the edge transitive closure constructed, the query time is constant $O(1)$.

In [8], Chen et al. propose an index by utilizing a spanning tree of the graph G. It takes $O(n+m)$ time to construct an index in $O(n+m)$ size. Given two nodes u and v in G, it can answer $u \rightsquigarrow v$ in $O(1)$ time if there is a path from u to v in the spanning tree, using a simple predicate, denoted $\mathcal{P}(,)$, between the codes (or labels) assigned to nodes over the spanning tree. We will discuss different encoding schema that assign codes (or labels) to nodes in G later in detail in this survey, and use codes and labels interchangeably. Let the codes for u and v be $\mathsf{code}(u)$ and $\mathsf{code}(v)$. If the predicate $\mathcal{P}(\mathsf{code}(u), \mathsf{code}(v))$ is true, then $u \rightsquigarrow v$ is true. However, because the codes are assigned based on the connections over the spanning tree of the graph G, it does not mean that $u \rightsquigarrow v$ is false if $\mathcal{P}(\mathsf{code}(u), \mathsf{code}(v))$ is false. There are edges in G that do not appear in the spanning tree. Chen et al. use an additional data structure called SSPI (Surrogate&Surplus Predecessor Index) to answer a reachability query in run time, which takes $O(m-n)$ time in the worst case. We call this approach Tree+SSPI. Like [8], a spanning tree of a graph G is also used in [32]. In [32], Trißl and Leser build an index, called GRIPP (GRaph Indexing based on Pre- and Postorder numbering), using a spanning tree of the graph G. Trißl and Leser discuss traversal strategies using the proposed GRIPP. The time and space complexities are the same to Tree+SSPI.

Wang et al. propose a dual-labeling approach in [34] for sparse graphs based on the observation that the majority of large graphs in real applications are sparse. It implies that the number of edges in the graph G that do not appear in a spanning tree of G is small. Let tree edges denote the edges that appear in the spanning tree, and non-tree edges denote the edges that do not appear in the spanning tree but appear in G. Let t be the number of such non-tree edges. Wang et al. consider to use a tree coding scheme (also called labeling) for tree edges and a graph coding (also called graph labeling) scheme for non-tree edges for sparse graphs where $t \ll n$. It handles the edge transitive closure over non-tree edges. The dual-labeling approach achieves $O(1)$ query time with an index of size $O(n+t^2)$ that is constructed in $O(n+m+t^3)$ time.

Agrawal et al. in [1] study a tree cover approach to assign labels to nodes in a DAG. In brief, if a node u can reach a node v, then u can reach any nodes in the subtree rooted at v. Agrawal et al. propose an optimal tree cover that maximally compresses the edge transitive closure. The index size is $O(n^2)$ in the worst case, but in practice, it can compress edge transitive closure which results in an even better compression rate than a chain cover [24, 9] which we

will discuss next. The time complexity for index construction is $O(nm)$. It can construct an index for a large graph efficiently. The query time is $O(\log n)$.

Jagadish in [24] proposes a chain cover approach. The chain cover is to decompose a graph G into pairwise disjoint chains. A chain is more general than a path. Consider a path $a \to b \to c \to d$ in G, where $x \to y$ represents a directed edge in G. The path can be considered as a chain itself, $a \rightsquigarrow b \rightsquigarrow c \rightsquigarrow d$, where $x \rightsquigarrow y$ represents y is reachable from x. The path can be decomposed into two pairwise disjoint chains, $a \rightsquigarrow c$ and $b \rightsquigarrow d$. Both $a \rightsquigarrow c$ and $b \rightsquigarrow d$ are not paths. Like the tree cover, if a node u can reach a node v, then u can reach any nodes in the chain from the position of the node v. Jagadish proposes an algorithm in $O(n^3)$ to find the minimal number of chains, in G. The number of chains for G is called the width of G, denoted by k. Based on the chain cover, an index in $O(nk)$ size can be constructed. The query time is $O(\log k)$. In [9], Chen and Chen propose a new approach that can further reduce the time complexity of constructing the index based on the chain over to $O(n^2 + kn\sqrt{k})$.

Jin et al. propose path-tree cover in [26] along the line of tree cover [1]. Jin et al. decompose G into pairwise disjoint paths and build a tree over the paths by treading a decomposed path as a node in the tree. Let k' be the number of pairwise disjoint paths in G. Two algorithms are proposed, namely, PTree-1 and PTree-2. Both construct an index in $O(nk')$ space. PTree-1 constructs the index in $O(nm)$ time, whereas PTree-2 constructs it in $O(mk')$ time. The query time is in $O(\log^2 k')$.

Cohen et al. in [17] propose an index called 2-hop cover. A node, u, in a graph G is assigned two sets of nodes, as its label, called $L_{in}(u)$ and $L_{out}(u)$. $L_{in}(u)$ contains a set of nodes that can reach u and $L_{out}(u)$ contains a set of nodes that u can reach. The labels assigned to nodes are done in a way to ensure $u \rightsquigarrow v$ to be true if and only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. It turns out to be a set cover problem. Cohen et al. propose an approximate algorithm to construct an index in $O(nm^{1/2})$ space. The time complexity for constructing such an index remains open. In [26], the conjecture is $O(n^3 \cdot |TC|)$ where $|TC|$ is the size of the edge transitive closure of G. Several efficient algorithms are proposed to compute 2-hop cover [29, 13, 14]. The 2-hop cover maintenance is studied in [30, 5]. Jin et al. in [25] further study a new approach, called 3-hop, that combines chain cover and 2-hop cover. The index construction time is $O(kn^2.|Con(G)|$. Here k is the number of pairwise disjoint paths in G, and $Con(G)$ is transitive closure contour of G defined in [25].

All the above are about how to answer reachability queries. Cohen et al. in [17] and Schenkel et al. in [30] address the distance-aware 2-hop cover which is to answer reachability queries with the shortest distance. Cheng and Yu in [10] propose efficient algorithms to fast compute distance-aware 2-hop cover.

The main difficult of computing distance-aware 2-hop cover is that it cannot condense a general directed graph into a DAG.

Before we discuss different graph coding schema, we explain a tree coding scheme for a tree. We call it single interval tree coding scheme in this survey. Many graph coding schema make use of the similar ideas used in the single interval tree coding scheme.

Single Interval Tree Coding Scheme: Let $G_S(V, E)$ be a tree. The single interval tree coding scheme (or simply SIT coding scheme) assigns a node $u \in G_S$ a code which is an interval, denoted $\mathsf{sitcode}(u) = [u_{start}, u_{end}]$, where u_{start} and u_{end} are two numbers such that $u_{start} < u_{end}$. The reachability, $u \rightsquigarrow v$, between two nodes, u and v, can be answered using the two corresponding codes, $\mathsf{sitcode}(u)$ and $\mathsf{sitcode}(v)$, in constant time $O(1)$. We denote it as a predicate $\mathcal{P}_{sit}(,)$

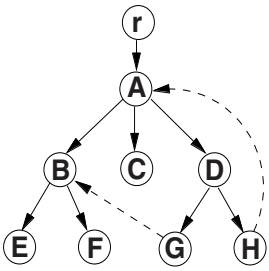
$$\mathcal{P}_{sit}(\mathsf{sitcode}(u), \mathsf{sitcode}(v)) = u_{start} < v_{start} \wedge v_{end} < u_{end}$$

Then, $u \rightsquigarrow v$ is true if and only if $\mathcal{P}_{sit}(\mathsf{sitcode}(u), \mathsf{sitcode}(v))$ is true. The codes can be assigned by traversing the tree G_S. Here, for a node, u, the u_{start} and u_{end} are the preorder and postorder values in a depth-first traversal of the tree. A counter is used with an initial value 0, and the counter value will increase by 1 before it visits another node in the traversal. In the tree traversal, a node will be visited twice. The u_{start} and u_{end} of a node u are assigned to be the counter values before and after all descendants of u have been traversed.

2. Traversal Approaches

In this section, we introduce two approaches, namely, Tree+SSPI [8] and GRIPP [32]. Both approaches use the SIT coding scheme to assign codes to nodes in a spanning tree of a graph G, and attempt to reduce the query processing time in traversal using either additional data structures or processing strategies. It is worth noting that Tree+SSPI [8] is proposed for pattern matching in a general context, and can be used to answer reachability queries.

Let $T_S(V_S, E_S)$ be a spanning tree of a graph $G(V, E)$. Here V_S and E_S are sets of nodes and edges of the spanning tree T_S. Note that $V_S = V$ and $E_S \subseteq E$. We use E_S to denote the set of tree edges of the graph G, and $E_R = E - E_S$ to denote the set of non-tree edges of the graph G that do not appear in E_S. In addition, below in discussions of Tree+SSPI and GRIPP, we assume that every node in G is assigned a code based on the SIT coding scheme. Given a reachability query $u \rightsquigarrow v$, Tree+SSPI and GRIPP first check whether the predicate $\mathcal{P}_{sit}(\mathsf{sitcode}(u), \mathsf{sitcode}(v))$ is true or not. If it is true, then $u \rightsquigarrow v$ is true. Otherwise, Tree+SSPI and GRIPP need to take additional actions to further check the reachability $u \rightsquigarrow v$, because u can reach v through a combination of tree edges and non-tree edges. Below, we discuss the cases that $u \rightsquigarrow v$ cannot be answered simply using the SIT coding scheme.



Node	*Start*	*End*	*Type*
r	0	21	tree
A	1	20	tree
B	2	7	tree
E	3	4	tree
F	5	6	tree
C	8	9	tree
D	10	19	tree
G	11	14	tree
B'	12	13	non-tree
H	15	18	tree
A'	16	17	non-tree

Figure 6.1. A Simple Graph G (left) and Its Index (right) (Figure 1 in [32])

2.1 Tree+SSPI

In [8], in addition to the SIT codes assigned to nodes, Chen et al. use another "space-economic" index, known as SSPI (Surrogate&Surplus Predecessor Index), to maintain information that needs to be used at run time to check reachability. The SSPI keeps a predecessor list for a node v in G, denoted as $PL(u)$. There are two types of predecessors. One is called *surrogate*, and the other is called *immediate surplus predecessor*. The two types of predecessors are explained in terms of the involvement of non-tree edges. Consider $u \rightsquigarrow v$ that must visit some non-tree edges on the path from u to v. Assume that (v_x, v_y) is the last non-tree edge on the path from u to v, then v_y is a surrogate predecessor of v if $v_y \neq v$ and v_x is an immediate surplus predecessor of v if $v_y = v$. SSPI can be constructed in a traversal of the spanning tree T_S of the graph G starting from the tree root. When a node v is visited, all its immediate surplus predecessors are added into $PL(v)$. Also, all nodes in $PL(u)$ are added into $PL(v)$, where u is the parent node of v in the spanning tree. It is sufficient to answer reachability queries using both SIT coding scheme and the SSPI.

To process a reachability query $u \rightsquigarrow v$, assuming that the SIT codes used return false when checking $u_{start} < v_{start} \wedge v_{end} < u_{end}$, Chen et al. design a *TwigStackD* algorithm. The *TwigStackD* algorithm checks the reachability via tree edges using run time stacks in traversing the spanning tree, and checks reachability via possible non-tree edges, using a partial solution pool that maintains some popped nodes from run time stacks temporally. The SSPI is used to answer which nodes can possibly reach a node v via non-tree edges.

2.2 GRIPP

Trißl and Leser in [32] use the SIT coding scheme in a different way. Instead of using SSPI and run time stacks, Trißl and Leser focus on how to traverse the

graph using the SIT codes. The graph dealt in [32] is a directed graph. We explain it using the same example used in [32]. Figure 6.1 shows a simple directed graph G on the left side and the GRIPP index table on the right side. The solid arrows indicate tree edges in G, and dotted arrows indicate non-tree edges in G. As shown in the GRIPP index table, a node in G is assigned with one or more than one SIT codes depending on the number of incoming edges to the node. The type in the GRIPP index table indicates the type of the incoming edge based on which the node is assigned a SIT code. The nodes with a type of non-tree in GRIPP index table are also called hop-nodes. Consider the node A, its SIT code, $\mathsf{sitcode}(A) = [A_{start}, A_{end}] = [1, 20]$, is assigned when A is traversed from/to r via the tree edge (r, A), and the duplication of A, a hop-node, denoted A', has a different SIT code $[16, 17]$, which is assigned when A is traversed from/to H via the non-tree edge (H, A). It can be understood that a directed graph G is represented as a tree with node duplications. In other words, all the hop-nodes, such as A' and B' in the GRIPP index table, are node duplications and become the leaf nodes in such a tree.

Trißl and Leser in [32] study how to reduce the traversing time when processing a reachability query. Consider $D \rightsquigarrow r$. Based on SIT codes given in the GRIPP index table, D can reach the nodes, G, H, A', and B', where A' and B' are two hop-nodes, because, $\mathsf{sitcode}(D) = [10, 19]$, $\mathsf{sitcode}(G) = [11, 14]$, $\mathsf{sitcode}(H) = [15, 18]$, $\mathsf{sitcode}(A') = [16, 17]$, and $\mathsf{sitcode}(B') = [12, 13]$. It implies that via the two hop-nodes, A' and B', there exists possibility that $D \rightsquigarrow r$ is true. Intuitively, it needs to hop to A and B to further traverse the graph G. Suppose it traverses A via the hop-node A' followed by traversing B via the hop-node B'. First, when it picks up A to traverse, it can traverse to A itself again, because A can reach H and then traverse to A via the hop-node A'. In this case, it does not need to traverse to A second time, because it cannot find any new possible reachability. Second, when it picks up B to traverse, it cannot find any new possible reachability, because A can reach B via tree edges and it has already explored all possible reachability via A that must include all the possible reachability via B. Based on the idea behind, Trißl and Leser study traversing order, pruning strategies, and and stop conditions. Because finding the optimal traversing order is NP-complete, Trißl and Leser propose some heuristics. For example, it attempts to traverse the giant strongly connected component first.

3. Dual-Labeling

Wang et al. in [34] investigate a dual-labeling coding scheme for a graph G. They use a SIT coding scheme to encode nodes that can be reached via tree edges over a spanning tree of the graph G, and a new coding scheme to encode nodes that can be possibly reached via non-tree edges. The codes assigned to

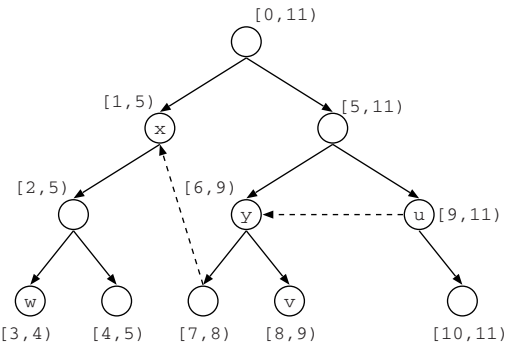


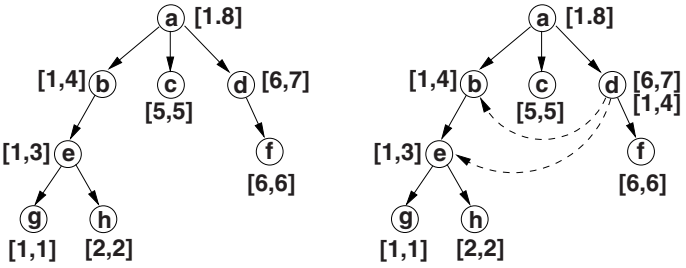
Figure 6.2. Tree Codes Used in Dual-Labeling (Figure 2 in [34])

nodes based on the tree edges over a spanning tree are slightly different from the SIT coding scheme used in GRIPP as seen in Figure 6.1. We also use the same example used in [34] to explain the main ideas.

Wang et al. assign modified SIT codes to nodes over a spanning tree of the graph G. We call it dual-tree code and denote it as $\mathsf{dtcode}(u)$ for $u \in G$, in the form of $[u_{start}, u_{end})$. An example is shown in Figure 6.2, where the solid arrows form a spanning tree and the dotted arrows are non-tree edges in G. The reachability $u \rightsquigarrow v$ over the spanning tree can be answered using $\mathsf{dtcode}(u)$ and $\mathsf{dtcode}(v)$ if $v_{start} \in \mathsf{dtcode}(u)$ is true. We give a predicate $\mathcal{P}_{dt}(,)$ to test whether $u \rightsquigarrow v$ is true over the spanning tree.

$$\mathcal{P}_{dt}(\mathsf{dtcode}(u), \mathsf{dtcode}(v)) = v_{start} \in \mathsf{dtcode}(u)$$

Note: it does not mean that u cannot reach v if $\mathcal{P}_{dt}(\mathsf{dtcode}(u), \mathsf{dtcode}(v))$ is false, because there exist other non-tree edges via which u can possibly reach v. In [34], a non-tree edge (u', v') is represented as $u'_{star} \rightarrow [v'_{start}, v'_{end})$ in a link table. Consider Figure 6.2, there are two non-tree edges, such that $9 \rightarrow [6, 9)$ and $7 \rightarrow [1, 5)$. The link table maintains the edge transitive closure over the non-tree edges and therefore is also called a transitive link table. For example, the existence of the two non-tree edges, $9 \rightarrow [6, 9)$ and $7 \rightarrow [1, 5)$, in the transitive link table implies that $9 \rightarrow [1, 5)$ exists in the transitive link table. It is because the node with the dtcode $[7, 8)$ can be reached from the node with the dtcode $[6, 9)$ and therefore the node with dtcode $[9, 11)$ can reach the node with dtcode $[1, 5)$. Let t be the number of non-tree edges, the transitive link table is in $O(t^2)$ space. A reachability query, $u \rightsquigarrow v$, can be answered using the transitive link table. Let $\mathsf{dtcode}(u) = [u_{start}, u_{end})$ and $\mathsf{dtcode}(v) = [v_{start}, v_{end})$. Then, $u \rightsquigarrow v$ is true if it can find an entry, $i \rightarrow [j, k)$, in the transitive link table such as $i \in [u_{start}, u_{end})$ and $v_{start} \in [j, k)$. The former implies that u can reach the non-tree edge and the latter implies that from the non-tree edge v can be reached.



(a) Tree Codes (b) Tree + Non-Tree Codes

Figure 6.3. Tree Cover (based on Figure 3.1 in [1])

In other to achieve $O(1)$ time, Wang et. al propose a transitive link count function (short for TLC function). As defined in Definition 1 in [34], *the proposed TLC function $N(x, y)$ computes the number of links $i \rightarrow [j, k)$ in the transitive link table that satisfy $i \geq x$ and $y \in [j, k)$*. Given two nodes, u and v, where $\mathsf{dtcode}(u) = [u_{start}, u_{end})$ and $\mathsf{dtcode}(u) = [u_{start}, u_{end})$. Assume that $\mathcal{P}_{dt}(\mathsf{dtcode}(u), \mathsf{dtcode}(t))$ is false. The following predicate $\mathcal{P}_{dg}(,)$ is defined over the graph via possible non-tree edges.

$$\mathcal{P}_{dg}(\mathsf{dtcode}(u), \mathsf{dtcode}(v)) = N(u_{start}, v_{start}) - N(u_{end}, v_{start}) > 0$$

$u \rightsquigarrow v$ is true over the possible non-tree edges if and only if the predicate $\mathcal{P}_{dg}(\mathsf{dtcode}(u), \mathsf{dtcode}(v))$ is true. Therefore, $u \rightsquigarrow v$ is true if and only if $\mathcal{P}_{dt}(\mathsf{dtcode}(u), \mathsf{dtcode}(v)) \vee \mathcal{P}_{dg}(\mathsf{dtcode}(u), \mathsf{dtcode}(v))$ is true.

Intuitively, it requires to maintain the TLC function $N(,)$ for every possible node pairs in G, which results in $O(n^2)$ space. In order to reduce it to $O(t^2)$ space, Wang et al. propose gridding and snapping techniques in [34]. Some techniques to trade off time for space are also discussed in [34].

4. Tree Cover

As an early work, in 1989, Agrawal et al. proposed a tree cover code. It uses multiple intervals to encode every node in a graph G. Consider a tree shown in Figure 6.3(a). A node u is assigned an interval $[u_{start}, u_{end}]$, where u_{end} is the postorder in traversing the tree, and u_{start} is the smallest postorder in the descendants of the subtree rooted at the node u. Like the other tree coding, $u \rightsquigarrow v$ is true over the tree, if and only if $v_{end} \in [u_{start}, u_{end}]$ is true. Agrawal et al. consider how to assign codes to nodes in DAG by inheriting codes from a node v to another node u if there is a non-tree edge (u, v) in the graph G. Consider the DAG shown in Figure 6.3(b). There are two additional non-tree edges (d, b) and (d, e). The node d will inherit $[1, 4]$ and $[1, 3]$ from the nodes b and e respectively. Because $[1, 3] \subseteq [1, 4]$, d only needs to have an additional interval $[1, 4]$. Therefore, the code for a node u in G, denoted as $\mathsf{tccode}(u) =$

Algorithm 1 Find-Tree-Cover(G)

1: let G' be a graph with an additional virtual root, γ, that links to all nodes in G that do not have any predecessors;
2: let L be the list of nodes in G' following a topological order;
3: $pred(\gamma) \leftarrow \emptyset$;
4: **for** each node v on L **do**
5: **for** each pair of incoming edges (u, v) and (u', v) **do**
6: **if** $|pred(u)| > |pred(u')|$ **then**
7: delete the edge (u', v);
8: **else**
9: delete the edge (u, v);
10: **end if**
11: **end for**
12: $pred(v) \leftarrow \{u\} \cup prev(u)$ for every incoming edge (u, v);
13: **end for**

$\{[u_{start_1}, u_{end_1}], [u_{start_2}, u_{end_2}], \cdots\}$, where u_{end_1} is the postorder when it traverses the spanning tree. In other words, $[u_{start_1}, u_{end_1}]$ is assigned to node u when traversing the spanning tree of the graph G, and the others are inherited from other nodes. Given the tree cover codes, $u \rightsquigarrow v$ is tree if and only if the postorder of v (v_{end_1}) is in an interval of the node u. The predicate $\mathcal{P}_{tc}(,)$ is given below.

$$\mathcal{P}_{tc}(\mathsf{tccode}(u), \mathsf{tccode}(v)) = \bigvee_i (v_{end_1} \in [u_{start_i}, u_{end_i}])$$

The total number of intervals for all codes in G becomes a factor to measure the quality of the tree cover. The total number varies depending on the selection of a spanning tree, known as tree cover, over the graph G. In [1], Agrawal et al. propose an algorithm to find the optimal tree cover. As shown in Algorithm 1, in order to achieve the optimal tree cover, for a node v, it retains the edge from the immediate predecessor of v with the maximum number of predecessors in the original DAG G, and delete the edges from the other immediate predecessors of v.

In [1], the storage issues and the tree-cover maintenance issue when a graph is updated are also discussed.

5. Chain Cover

Jagadish [24] proposes a chain cover coding scheme to answer a reachability query on a DAG G. A chain cover of G is a set of pairwise disjoint chains, $C_1, C_2, \cdots, C_k$. Here, a chain $C_i = v_{i_1} \rightsquigarrow v_{i_2} \rightsquigarrow \cdots \rightsquigarrow v_{i_k}$ where v_{i_j} is a node in G and $v_{i_{j+1}}$ is reachable from v_{i_j} in G. The union of the nodes in

Algorithm 2 Compute-Chain-Cover(G, $\{C_1, C_2, \cdots, C_k\}$)

Input: The DAG G, and a chain cover $\{C_1, \cdots, C_k\}$
Output: The chain cover code for every node in G

1: sort all nodes in G in topological order;
2: let every node v_i in G unmarked;
3: **while** there are unmarked node v_i in G that do not have unmarked immediate successors **do**
4: $\mathsf{chaincode}(v_i) \leftarrow \{(1,\infty),(2,\infty),\cdots,(k,\infty)\}$;
5: let $L_{i,x}$ denote the x-th pair in $\mathsf{chaincode}(v_i)$;
6: let $suc(v_i)$ denote the immediate successors of v_i in G;
7: **for** every $v_j \in suc(v_i)$ **do**
8: **for** $l = 1$ **to** k **do**
9: $(l, p_{j,l}) \leftarrow L_{j,l}$;
10: $(l, p_{i,l}) \leftarrow L_{i,l}$;
11: **if** $p_{j,1} \leq p_{i,l}$ **then**
12: $L_{i,l} \leftarrow (l, p_{j,l})$;
13: **end if**
14: **end for**
15: **end for**
16: mark v_i;
17: **end while**
18: **return** the set of $\mathsf{chaincode}(v_i)$ for every $v_i \in G$;

all chains is the entire set of nodes in G, and the intersection of nodes in any two chains is empty. The optimal chain cover of G is a chain cover of G that contains the least number of chains among all possible chain covers of G.

Suppose the chain cover contains k chains, to answer the reachability queries, each node $v_i \in G$ is assigned a code, denote $\mathsf{chaincode}(v_i)$, which is a list of pairs, $\{(1, p_{i,1}), (2, p_{i,2}), \cdots, (k, p_{i,k})\}$. Each pair $(j, p_{i,j})$ means that the node v_i can reach any nodes from the position $p_{i,j}$ in the j-th chain. If v_i cannot reach any node in the j-th chain, then $p_{i,j} = +\infty$. The chain cover index contains $\mathsf{chaincode}(v_i)$ for every node v_i in G.

A reachability query $v_a \rightsquigarrow v_d$ can be answered using a predicate $\mathcal{P}_c(,)$ such that $v_a \rightsquigarrow v_d$ is true if and only if v_a appears at the $p_{a,j}$ position in a chain C_j and $p_{d,j} \leq p_{a,j}$. In other words, v_a can reach v_d in a chain C_j. All pairs in the chain cover index for G can be indexed and stored using a B+-tree. Answering a reachability query needs $O(\log(n))$ time with $O(n \cdot k)$ space.

Given a chain cover $C_1, C_2, \cdots, C_k$ of a DAG G, Algorithm 2 shows how to compute $\mathsf{chaincode}(v_i)$ for every $v_i \in G$. It visits every node in G in the reverse of topological order (line 3). For each node visited, its $\mathsf{chaincode}(v_i)$ is updated using its immediate successors if the corresponding position in the l-th

chain, C_l, of an immediate successor is smaller than the current position v_i has in C_l. Let d_i be the out degree of node v_i (the number of immediate successors of v_i). The time complexity of Algorithm 2 is $O(\sum_{i=1}^{n}(d_i \cdot k)) = O(mk)$, where m is the number of edges in G. It becomes important to make k as small as possible. Below, we introduce two approaches that aim at computing the optimal chain cover with the minimal k.

5.1 Computing the Optimal Chain Cover

Jagadish in [24] proposes a min-flow approach to compute the optimal chain cover of a DAG G. The main idea is as follows. It constructs another graph H. For every node $v_i \in G$, it adds two nodes, x_i and y_i, in H and a directed edge (x_i, y_i) in H. In other words, a node in G is represented as an edge in H. For each edge (v_i, v_j) in G, it adds an edge (y_i, x_j) in H. A source node is added into H that links to every node with in-degree 0 in H, and a sink node is added that is linked by every node with out-degree 0 in H. Then, Jagadish proposes to find the min-flow from the source node to the sink node such that every edge (x_i, y_i) has a positive flow. It can be solved in time $O(n^3)$. Here, each flow corresponds to a chain in G. In such a way, it can get the chain cover of G. If a node may appear in several chains, it keeps one occurrence in any chain and removes the other occurrences.

Chen and Chen in [9] propose an approach using bipartite matching. All nodes in the DAG G are decomposed into several layers, $V_1, V_2, \cdots, V_h$, where h is the length of the longest path in G. The layers can be constructed as follows. V_1 is the set of nodes with out-degree 0 in G, and V_i is the set of nodes with out-degree 0 when the nodes in V_k, for $1 \leq k < i$ are removed from G. This can be done in $O(m)$ time.

Algorithm 3 shows how to find the optimal chain cover based on the layers. The main idea of Algorithm 3 is as follows. In each successive layers, it finds the maximum matching for the bipartite graph induced by the nodes in the two layers (line 1-4). For some unmatched node v, it adds a virtual node v' in the top of the two successive layer, in order to be further matched by nodes in the unseen upper layers (line 5-9). A potential edge (u, v') for some $u \in V_{i+2}$ is added, if and only if there is an edge from u to a node $x \in V_{i+1}$ and there is an alternating path from x to v'. A path is alternating with respect to M_i if and only if its edges alternately appear in $E_i \setminus M_i$ and M_i, where M_i is the maximum matching of the bipartite graph and E_i is the bipartite graph in the i-th iteration. Then, in line 10-13, each virtual node is resolved using the alternating paths by removing the virtual nodes, transferring the edges in the alternating paths, and adding the new edge from u to x as discussed above. An example for resolving a virtual node v' by an alternating path is illustrated in Figure 6.4. The optimal chain cover can be computed in time $O(n^2 + kn\sqrt{k})$

Algorithm 3 Optimal-Chain-Cover(G, $\{V_1, V_2, \cdots, V_h\}$)

Input: a DAG G, and the layers $V_1, \cdots, V_h$
Output: The optimal chain cover $C_1, \cdots, C_k$

1: $V_1' \leftarrow V_1$;
2: **for** $i = 1$ **to** $h - 1$ **do**
3: $V_{i+1}' \leftarrow V_{i+1}$;
4: $M_i \leftarrow$ maximum matching of the bipartite graph induced by V_i' and V_{i+1}';
5: **for all** unmatched node $v \in V_i'$ in M_i **do**
6: create a virtual node v' in G;
7: $V_{i+1}' \leftarrow V_{i+1}' \cup \{v'\}$;
8: $M_i \leftarrow M_i \cup (v', v)$;
9: create potential edges (u, v') for some $u \in V_{i+2}$;
10: **end for**
11: **end for**
12: $CH \leftarrow M_1 \cup M_2 \cup \cdots \cup M_h$;
13: **for** $i = 1$ **to** $h - 1$ **do**
14: **for all** virtual node $v' \in V_i'$ **do**
15: resolve v' from CH using alternating paths in M_i;
16: **end for**
17: **end for**
18: **return** CH;

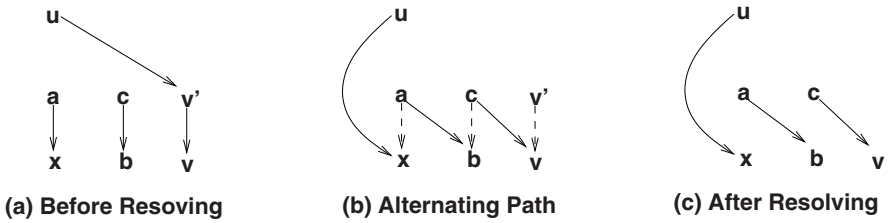


Figure 6.4. Resolving a virtual node

where n is the number of nodes in G and k is the number of chains in the optimal chain cover (known as the width of G).

6. Path-Tree Cover

Jin et al. in [26] propose a path-tree cover coding scheme to answer a reachability query on a DAG $G(V, E)$.

First, the graph $G(V, E)$ is decomposed into a set of pairwise disjoint paths, $P_1, P_2, \cdots, P_{k'}$. Here, a path $P_i = v_{i_1} \rightarrow v_{i_2} \rightarrow \cdots \rightarrow v_{i_k}$ where $v_{i_j} \rightarrow v_{i_{j+1}}$ is an edge in G. A path cover consists of k' paths such that (a) the union of

the nodes in all the paths is the entire set of nodes in G and (b) the intersection of two paths is empty. The optimal path cover of G is a path cover of G that contains the least number of paths among all possible path covers of G. Such optimal path cover can be obtained using Simon's algorithm in [31].

Second, let P_i and P_j be two paths computed in the path cover. There may exist edges from some nodes in P_i to some nodes in P_j, denoted as $E_{P_i \to P_j}$, which is a subset of the edges in G. Some edges in $E_{P_i \to P_j}$ can be eliminated losslessly. For example, suppose $P_i = w$ and $P_j = u \to v$, and assume $E_{P_i \to P_j}$ consists of two edges from P_i to P_j, $\{w \to u, w \to v\}$. Then $w \to v$ can be eliminated, because there is a path $w \to u \to v$ that can answer the reachability query $w \rightsquigarrow v$. The similar can be done if there are edges from P_j to P_i in reverse order. The edge elimination in this way is lossless because it does not lose any reachability information. Let $E'_{P_i \to P_j}$ be a subset of $E_{P_i \to P_j}$ after edge elimination. Jin et al. show that all edges in $E'_{P_i \to P_j}$ are in parallel. Furthermore, Jin et al. use a single weighted edge from P_i to P_j, in order to represent how many nodes in P_i can reach a node in P_j. Based on the weighted edges from P_i to P_j, a weighted path-graph $G_P(V, E)$ is constructed. Here, V is a set of nodes representing paths, $P_1, P_2, \cdots, P_{k'}$, computed in the path cover, and E is a set of edges (P_i, P_j) with a weight, if $E'_{P_i \to P_j} \neq \emptyset$.

Third, based on the path-graph $G_P(V, E)$, Jin et al. construct a spanning tree $T_P(V, E)$, called path-tree, with two criteria: MaxEdgeCover and MinPathIndex. The former means to cover as many edges in G as possible, and the latter means to reduce the size of a resulting path-tree cover as much as possible. The path tree is computed using the algorithm presented in [16, 21].

Finally, a path-tree cover code, $\mathsf{ptcode}(u)$, is assigned to node $u \in G$ based on the path-tree T_P. The $\mathsf{ptcode}(u) = ((u_{start}, u_{end}), (u_x, u_y))$ consists of two pairs. The first pair is the interval $[u_{start}, u_{end}]$, like SIT code, assigned to the path P_i where u resides uniquely, because a node represents a path in T_P. The second pair (u_x, u_y) is used to record the position of the node u in the path P_i. A reachability query, $u \rightsquigarrow v$ is answered to be true, if the predicate $\mathcal{P}_{pt}(\mathsf{ptcode}(u), \mathsf{ptcode}(v))$ is true, such as $[v_{start} v_{end}] \subset [u_{start}, u_{end}] \wedge u_x < v_x \wedge u_y < u_y$. It is important to note that it does not mean $u \rightsquigarrow v$ is false if $\mathcal{P}_{pt}(\mathsf{ptcode}(u), \mathsf{ptcode}(v))$ is false, because the path-tree cover code and the predicate are both defined over the path-tree T_P. There may exist edges that cannot be fully covered by the path-tree.

The path-tree cover coding scheme is different from the tree cover [1] and the chain cover [24, 9]. Both tree cover and chain cover coding schema answer reachability queries only using the predicates, $\mathcal{P}_{tc}(,)$ and $\mathcal{P}_c(,)$, respectively. On the other hand, the path-tree cover coding scheme cannot answer reachability queries only using the predicate $\mathcal{P}_{pt}(,)$. The path-tree cover coding scheme shares similarity with the dual-labeling [34], and aims at covering as many non-tree edges as possible. Jin et al. in [26] show that the path-tree cover is

superior over the optimal tree cover [1] and optimal chain cover [24] in terms of the compression ability.

7. 2-HOP Cover

Cohen et al. propose a 2-hop cover in [17] for a graph G. In a 2-hop cover, a node in G is assigned to a 2-hop code, $\mathsf{2hopcode}(u) = (L_{in}(v), L_{out}(v))$, where $L_{in}(v)$ and $L_{out}(v)$ are subsets of the nodes in G. Based on the 2-hop cover, a reachability query $u \rightsquigarrow v$ is to be answered true if and only if $\mathcal{P}_{2hop}(\mathsf{2hopcode}(u), \mathsf{2hopcode}(v))$ is true.

$$\mathcal{P}_{2hop}(\mathsf{2hopcode}(u), \mathsf{2hopcode}(v)) = L_{out}(u) \cap L_{in}(v) \neq \emptyset$$

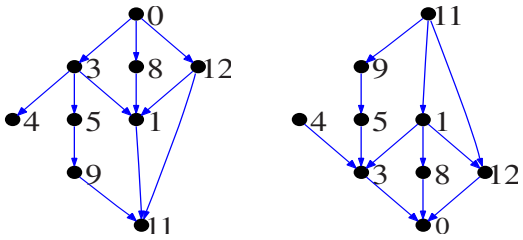
The main idea behind 2-hop cover coding scheme is to compress the edge transitive closure of G. Let $TC(G)$ be the edge transitive closure of G. A pair (u, v) in $TC(G)$ indicates that $u \rightsquigarrow v$ is true in G. Consider a node w in G as a center. All the ancestors of w, denoted as $ancs(w)$, can reach w, and w can reach any of its descendants, denoted as $desc(w)$. In other words, $ancs(w)$ is the set of nodes $\{u\}$ if $(u, w) \in TC(G)$ and $desc(w)$ is the set of nodes $\{v\}$ if $(w, v) \in TC(G)$. Let $A_w \subseteq ancs(w) \cup \{w\}$ and $D_w \subseteq desc(w) \cup \{w\}$. A complete bipartite graph, called a 2-hop cluster, is denoted $S(A_w, w, D_w)$, with the center w. A 2-hop cluster $S(A_w, w, D_w)$ indicates that every node, u in A_w can reach any node v in D_w, or $u \rightsquigarrow v$ is true for every $u \in A_w$ and $v \in D_w$. Given a cluster $S(A_w, w, D_w)$, it implies that if w is added into $L_{out}(u)$ for every $u \in A_w$ and is added into $L_{in}(v)$ for every $v \in D_w$, the reachability information presented by the complete bipartite graph $S(A_w, w, D_w)$ is completely preserved, because $u \rightsquigarrow v$ is true if and only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. A $S(A_w, w, D_w)$ compactly represents $|A_w| \cdot |D_w| - 1$ pairs in $TC(G)$ in total with a space cost of $|A_w| + |D_w|$. A 2-hop cover is a set of 2-hop clusters that completely covers the edge transitive closure $TC(G)$.

The optimal 2-hop cover problem is to find the minimum size 2-hop cover, which is proved to be NP-hard [17]. Based on the greedy algorithm for minimum set cover problem [27], Cohen et al. give an approximation algorithm to get a nearly optimal 2-hop cover which is larger than the optimal one at most $O(\log n)$.

Algorithm 4 illustrates the ideas [17]. It computes the edge transitive closure $TC(G)$ (line 1). Let TC' be $TC(G)$ (line 2). In every iteration, it finds a 2-hop cluster $S(A_w, w, D_w)$ that has the maximum ratio, $(|S(A_w, w, D_w) \cap TC'|)/(|A_w| + |D_w|)$, among all possible 2-hop clusters. Here, TC' is used to indicate the set of pairs in $TC(G)$ that are not covered by any 2-hop clusters computed yet. After identifying the $S(A_w, w, D_w)$ with the maximum ratio in the current iteration, it removes all the pairs (u, v) from TC' if $u \in A_w$ and $v \in D_w$ (line 5). In line 6-7, it updates 2-hop cover codes.

Algorithm 4 2Hop-Cover(G)

1: compute the edge transitive closure $TC(G)$ of G;
2: $TC' \leftarrow TC(G)$;
3: **while** $TC' \neq \emptyset$ **do**
4: find the max $S(A_w, w, D_w)$;
5: remove all the pairs in TC' that are covered by $S(A_w, w, D_w)$;
6: add w into $L_{out}(u)$ if $u \in A_w$;
7: add w into $L_{in}(v)$ if $v \in D_w$;
8: **end while**



(a) $G_{\downarrow}(V_{\downarrow}, E_{\downarrow})$ (b) $G_{\uparrow}(V_{\uparrow}, E_{\uparrow})$

Figure 6.5. A Directed Graph, and its Two DAGs, $G_{\downarrow}$ and $G_{\uparrow}$ (Figure 2 in [13])

The computational cost is high as can be seen in Algorithm 4. First, it needs to compute the edge transitive closure. Second, it needs to rank all 2-hop clusters $S(A_w, w, D_w)$ based on $(|S(A_w, w, D_w) \cap TC'|)/(|A_w| + |D_w|)$ in every iteration. Third, it is difficult to compute 2-hop cover for a large graph.

7.1 A Heuristic Ranking

Schenkel et al. in [29] propose a heuristic ranking to avoid to recompute and rank all $(|S(A_w, w, D_w) \cap TC'|)/(|A_w| + |D_w|)$ for all possible centers $S(A_w, w, D_w)$ in every iteration. The idea is as follows. It computes all $|S(A_w, w, D_w) \cap TC'|/(|A_w| + |D_w|)$, for all nodes in G. Initially, $TC' = TC(G)$. Let d_w denote $|S(A_w, w, D_w) \cap TC'|/(|A_w| + |D_w|)$. It initially maintains all the pairs of (w, d_w) in a priority queue. The first is with the max ratio d_w value. In every iteration, it picks up the first (w, d_w) and recomputes $d'_w = |S(A_w, w, D_w) \cap TC'|/(|A_w| + |D_w|)$, if $d_w > d'_w$, the pair (w, d'_w) is enqueued into the priority queue. It repeats until it picks a node w such that $d_w = d'_w$. In practice, Schenkel et al. find that it only needs to repeat 2-3 times in every iteration on average.

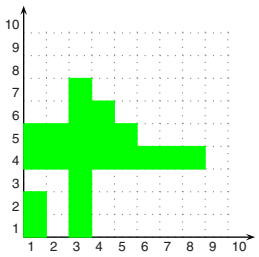


Figure 6.6. Reachability Map

w	tccode(w) for $w \in G_{\downarrow}$		tccode(w) for$w \in G_{\uparrow}$	
	$po_{\downarrow}(w)$	$I_{\downarrow}(w)$	$po_{\uparrow}(w)$	$I_{\uparrow}(w)$
0	9	[1,9]	4	[4,4]
1	1	[1,1],[3,3]	3	[1,5]
3	6	[1,6]	5	[4,5]
4	2	[2,2]	9	[4,5],[9,9]
5	5	[3,5]	6	[4,6]
8	7	[1,1],[3,3],[7,7]	1	[1,1],[4,4]
9	4	[3,4]	7	[4,7]
11	3	[3,3]	8	[1,8]
12	8	[1,1],[3,3],[8,8]	2	[2,2],[4,4]

Table 6.2. A Reachability Table for $G_{\downarrow}$ and $G_{\uparrow}$

7.2 A Geometrical-Based Approach

Cheng et al. in [13] propose a geometrical-based approach that does not need to compute the edge transitive closure of $TC(G)$ directly, and speeds up the computing of max ratio of the 2-hop clusters using an R-tree, in particular for a large dense graph G.

First, instead of computing the edge transitive closure $TC(G)$, Cheng et al. compute tree cover [1], because in practice the tree cover algorithm in [1] is very fast. The tree cover codes are used to compute 2-hop cover. Consider Figure 6.5(a) which shows a DAG $G_{\downarrow}(V_{\downarrow}, E_{\downarrow})$. Suppose it needs to assign 2-hop codes to the graph shown in Figure 6.5(a). Cheng et al. compute the tree cover codes for $G_{\downarrow}(V_{\downarrow}, E_{\downarrow})$, and compute the tree cover codes for another corresponding graph $G_{\uparrow}(V_{\uparrow}, E_{\uparrow})$, which is a graph that by changing every edge $(u, v) \in G_{\downarrow}$ to (v, u). The Table 6.2 shows the tccode(w) for the node w in

$G_\downarrow$ and $G_\uparrow$. In particular, $po_\downarrow(w)$ and $po_\uparrow(w)$ indicate the postorder of w, and $I_\downarrow(w)$ and $I_\uparrow(w)$ indicate the intervals of w, in $G_\downarrow$ and $G_\uparrow$, respectively.

Second, based on the tree cover codes, Cheng et al. construct a 2-dimensional reachability map, a node w is mapped onto the (x_w, y_w) position in the reachability map as $(po_\downarrow(w), po_\uparrow(w))$. The reachability information $u \rightsquigarrow v$ is mapped onto 2-dimensional reachability map, (x_v, y_u). If $u \rightsquigarrow v$ is true, then $(x_v, y_u) = 1$, otherwise $(x_v, y_u) = 0$. Therefore, the same reachability information, that a 2-hop cluster $S(A_w, w, D_w)$ represents, is represented as a number of rectangles in the 2-dimensional reachability map.

With the assistance of the 2-dimensional reachability map, Cheng et al. find the max $S(A_w, w, D_w)$ in line 4 of Algorithm 4 as to find the max coverage of rectangles, which can be done using an R-tree. It is important to note that Cheng et al. in [13] try to maximize $|S(A_w, w, D_w) \cap TC'|$ instead of $|S(A_w, w, D_w) \cap TC'|/(|A_w| + |D_w|)$. Both are set cover problems.

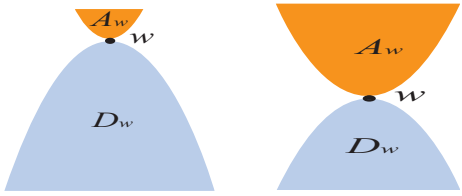
7.3 Graph Partitioning Approaches

In this section, we discuss three graph partitioning approaches used in computing a 2-hop cover for a large graph G.

A Flat Partitioning Approach. Schenkel et al. propose a flat partitioning approach in [29] to compute 2-hop cover in three steps. First, it partitions the graph G into k subgraphs $G_1, G_2, \cdots, G_k$ depending on the available memory M. Second, it computes the edge transitive closure and the 2-hop cover for each subgraph G_i, for $1 \leq i \leq k$, using Algorithm 4 with the heuristic ranking discussed in the previous subsection. Third, it merges the k 2-hop covers computed for the k subgraphs, $G_1, G_2, \cdots, G_k$, by dealing with the edges that cross subgraphs. It is called a cover joining step, and the cover joining yields a 2-hop cover for the entire graph G. The cover joining is done as follows. Suppose the 2-hop covers for all k subgraphs are computed. Let (u, v) be a cross-partition edge where $u \in G_i$ and $v \in G_j$ and $G_i \neq G_j$. Schenkel et al. compute the 2-hop cover for G by encoding all reachability via (u, v) according to the following two operations.

- For all $a \in ancs(u)$, $L_{out}(a) \leftarrow L_{out}(a) \cup \{u\}$, and
- For all $d \in desc(v) \cup \{v\}$, $L_{in}(d) \leftarrow L_{in}(d) \cup \{u\}$.

It means that, 2-hop clusters, $(ancs(u), u, desc(u))$, for all cross-partition edges (u, v), are covered mandatorily to encode G. The compression rate of $TC(G)$ using the flat partitioning decreases. As reported in [29, 30], the cover joining becomes the bottleneck of the whole processing. Schenkel et al. in [30] propose an effective and efficient approach for the third step of cover joining, using a skeleton graph (SG).



(a) Unbalanced (b) Balanced

Figure 6.7. Balanced/Unbalanced $S(A_w, w, D_w)$

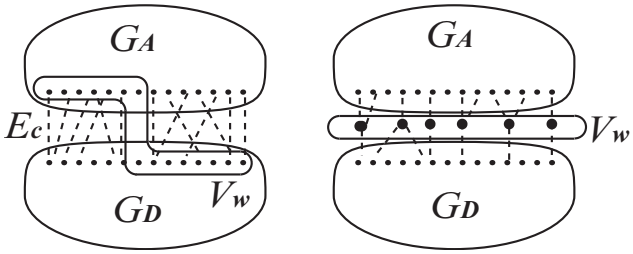
A skeleton graph is constructed at the partition-level. Suppose a graph $G(V, E)$ is partitioned into k subgraphs $G_1(V_1, E_1)$, $G_2(V_2, E_2)$, $\cdots$, $G_k(V_k, E_k)$. Here, $V = \cup_{i=1}^{k} V_i$ and $V_i \cap V_j = \emptyset$ if $i \neq j$. $E = E_C \cup (\cup_{i=1}^{k} E_i)$ where $E_i \cap E_j = \emptyset$ if $i \neq j$ and E_C is the set of cross-partition edges $E \setminus (\cup_{i=1}^{k} E_i)$. The skeleton graph $G_S(V_S, E_S)$ is constructed as follows. Here, V_S is a set of nodes u if u appears in a cross-partition edge in E_C. E_S contains all the cross-partition edges E_C, and in addition contains edges that explicitly indicate whether two cross-partition edges are connected via some paths in a subgraph. Consider a subgraph G_i, and let (v_i, v_j) and (v_k, v_l) be any two cross-partition edges such that v_j and v_k as nodes appear in G_i. There will be an edge (v_j, v_k) in E_S if $v_j \rightsquigarrow v_k$ is true in G_i. Schenkel et al. compute a 2-hop cover for G_S using Algorithm 4 with the heuristic ranking. At this stage, for a node $u \in G$ that does not appear in any cross-partition edges, u has a $\mathsf{2hopcode}(u)$ which is computed in G_i where u resides. For a node $u \in G$ that appears in cross-partition edges, it has two 2-hop cover codes. One is computed because it appears in a subgraph G_i, $\mathsf{2hopcode}(u)$. The other is the one computed in the skeleton graph G_S, denoted $\mathsf{2hopcode}'(u)$. Let $\mathsf{2hopcode}(u) = (L_{in}(u), L_{out}(u))$ and $\mathsf{2hopcode}'(u) = (L'_{in}(u), L'_{out}(u))$.

The final 2-hop cover code is computed by augmenting the 2-hop cover code computed for G_i using the 2-hop cover code computed over the skeleton graph. Let (u, v) be a cross-partition edge, where $u \in G_i$ and $v \in G_j$, and let $V(G_i)$ and $V(G_j)$ denote the sets of nodes in G_i and G_j. It is done using the following two operations.

- For all $a \in ancs(u) \cap V(G_i)$, $L_{out}(a) \leftarrow L_{out}(a) \cup L'_{out}(u)$, and
- For all $d \in desc(v) \cap V(G_j)$, $L_{in}(d) \leftarrow L_{in}(d) \cup L'_{in}(v)$.

The skeleton graph gives a global picture over the 2-hop cover and can compress the edge transitive closure effectively.

A Hierarchical Partitioning Approach. Cheng et al. in [14] consider the quality of the partitioning. The partitioning divides a large graph into smaller graphs and computes the 2-hop cover code for the large graph by augmenting



(a) Node-Oriented (b) Edge-Oriented

Figure 6.8. Bisect G into G_A and G_D (Figure 6 in [14])

the 2-hop cover codes for smaller graphs. The main issue in the flat partitioning [29, 30] is to find a way to compute 2-hop cover codes for a large graph with the limited memory. Because it is not easy to find an optimal partitioning of graphs, Schenkel et al. take a simple approach. For a DAG graph G, it can start from the top or the bottom (refer to $G_{\downarrow}$ in Figure 6.5) to extract a subgraph that can be held in memory, and repeats it until the entire graph is decomposed into a set of smaller graphs. Consider a node w appearing in a cross-partition edge. The node w has potential power to compress the edge transitive closure effectively, because many nodes in one subgraph may connect to many nodes in another subgraph via the node w. However, there are two cases as illustrated in Figure 6.7. The flat partitioning may result a partitioning that result in many unbalanced 2-hop clusters $S(A_w, w, D_w)$ (Figure 6.7(a)). Cheng et al. attempt to partition a graph that results in balanced 2-hop clusters $S(A_w, w, D_w)$ (Figure 6.7(b)). Recall $S(A_w, w, D_w)$ uses $|A_w| + |D_w|$ space to compress $|A_w| \cdot |D_w| - 1$ entries in the edge transitive closure. Cheng et al. show that the compression rate $(|A_w| \cdot |D_w| - 1)/(|A_w| + |D_w|)$ is maximum when $|A_w| = |D_w|$.

Cheng et al. in [14] propose a hierarchical partitioning approach to partition a large graph G into two subgraphs, G_A and G_D, repeatedly in a top-down fashion. It repeats if a subgraph cannot be held in memory in such a manner.

The key idea presented in [14] is to select a set of centers, $V_w = \{w_1, w_2, \cdots\}$, as a cut to partition a graph G. Note that the set of centers implies a set of 2-hop clusters, $S(A_{w_1}, w_1, D_{w_1})$, $S(A_{w_2}, w_2, D_{w_2})$, $\cdots$. Suppose that G is partitioned into G_A and G_D. There exist a set of edges (u, v) where $u \in G_A$ and $v \in G_D$. Let E_C denote such a set of edges. Cheng et al. propose a node-oriented and an edge-oriented approach to identify V_w where $w_i \in V_w$ is selected from the set of nodes appearing in E_C. As illustrated in Figure 6.8(a), in the node-oriented approach, it selects a set of nodes in E_C as V_w. As illustrated in Figure 6.8(b), in the edge-oriented approach, it treats edges as virtual nodes and identify V_w. The set of V_w is computed as to find the

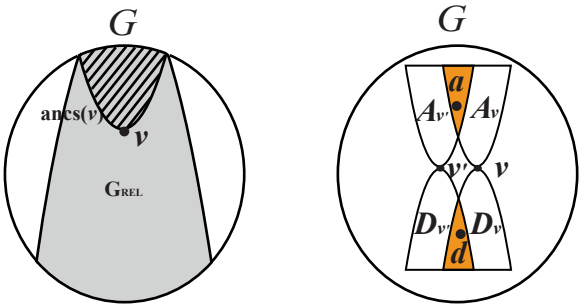
minimum 2-hop cover to cover reachability cross G_A and G_D from the nodes appearing in E_C. It is important to note that reachability between the two subgraphs, G_A and G_D, are completely covered by the set of 2-hop clusters using the set of nodes V_w. Based on V_w, Cheng et al. extract an induced subgraph of G_A, denoted $G_\top$, which does not include any nodes in V_w, and extract an induced subgraph of G_D, denoted $G_\bot$, which does not include any nodes in V_w. Both $G_\top$ and $G_\bot$ are treated as G in the next steps to bisect.

7.4 2-Hop Cover Maintenance

A 2-hop cover is hard to compute. Schenkel et al. in [30] and Bramandia et al. in [5] study the 2-hop cover maintenance problem to minimize the effort of updating the 2-hop cover when updates occur, and avoid computing a 2-hop cover from the beginning. There are four operations, insertion/deletion of nodes/edges. It is straightforward to deal with insertions. Consider an insertion of a new edge between an existing node and a new node v to G. A simple solution is to insert $S(ancs(v), v, desc(v))$ into the 2-hop cover, i.e., inserting v to the L_{in} and L_{out} of all nodes in $desc(v)$ and $ancs(v)$, respectively. The deletion of nodes/edges becomes non-trivial, because a deletion of a node w may affect the reachability $u \rightsquigarrow v$ if $w \in L_{out}(u)$ and $w \in L_{in}(v)$. Removing w from $L_{out}(u)$ and $L_{in}(v)$ may make $u \rightsquigarrow v$ to be wrongly answered as false, because there may be other paths from u to v. The existing work focus on deletion operations. In this article, we mainly discuss their approaches to handle the deletion of an existing node. The similar idea can be applied to handling the deletion of an existing edge.

Re-labeling a subgraph. When there is a deletion of an existing node, Schenkel et al. in [30] compute a 2-hop cover $\hat{L}$ of a subgraph G_{REL} of G, in order to reflect all the affected connections in G, due to the deletion of an existing node v. The existing 2-hop cover L for the graph G, before updating, will be updated to reflect all the affected connections by incorporating $\hat{L}$. The graph $G_{\text{REL}}(V_{\text{REL}}, E_{\text{REL}})$ is constructed as an induced graph of G, denoted as $G[V_{\text{REL}}]$. The set of nodes, V_{REL} is computed as follows. First, it includes all nodes in $ancs(v)$ in V_{REL}, which is shown as the striped region in Figure 6.9a. Second, it includes all nodes in $desc(u)$ into V_{REL} if $u \in ancs(v)$, which is shown as the gray region in Figure 6.9a. Note that G_{REL} represents all the affected connections.

The 2-hop cover $\hat{L}$ computed for G_{REL} is used to update the 2-hop cover L for the entire graph G as follows. It is obvious that all the connections (a, d), that exist in G, need to be updated if $a \in V_{\text{REL}}$. Note that $d \in V_{\text{REL}}$ in this case. All $L_{out}(a)$ for $a \in V_{\text{REL}}$ are updated as to be $\hat{L}_{out}(a)$. On the other hand, for a connection (a, d) that exists in G where $d \in V_{\text{REL}}$, the node a may or may not



(a) Re-labeling a subgraph (b) Reserving alternative paths

Figure 6.9. Two Maintenance Approaches

exist in V_{REL}. If $a \in V_{\text{REL}}$, $\hat{L}_{in}(d)$ are used to reflect all (a, d), because a and d are both in G_{REL}. For the latter case, it keeps $L_{in}(d) \setminus V_{\text{REL}}$, because such (a, d) are not affected by the deletion of v and are encoded by previous 2-hop clusters. Hence, $L_{in}(d)$ is updated as $(L_{in}(d) \setminus V_{\text{REL}}) \cup \hat{L}_{in}(d)$.

A drawback of this approach is high maintenance cost, because G_{REL} can be as large as G itself. It means that the maintenance for the current 2-hop cover degrades into the re-computation of a new 2-hop cover for the entire graph. Bramandia et al. [4] show the 2-hop cover code maintenance using the geometrical-based approach [13].

Reserving all alternative paths. Bramandia et al. in [5] propose u2-hop that can work on a smaller set of affected connections online at the expense of a large space. It considers all connections (a, d), where $a \in ancs(v)$ and $d \in desc(v)$, and modifies $L_{out}(a)$ and $L_{in}(d)$ by removing (i) v, (ii) nodes that are on longer reachable from a or nodes that can not reach d any longer, due to the deletion of the node v. The operation (i) is to exclude $S(A_v, v, D_v)$ from the current 2-hop cover. The operation (ii) is to maintain $S(A_w, w, D_w)$, where $w \in ancs(v)$ or $w \in desc(v)$, by removing those nodes in A_w and D_w which no longer connect to w. In order to maintain the 2-hop cover, it is important to note that the succinct maintaining operations of [5] require redundancy in the 2-hop cover. Such redundancy comes from the requirement that for any connection (a, d) in G, it repeatedly encodes it with multiple 2-hop clusters for all different alternative paths from a to d, as illustrated by Figure 6.9b. The example shows that two alternative paths from a to d exist in G, and v and v' are contained in the two paths respectively. So both $S(A_v, v, D_v)$ and $S(A_{v'}, v', D_{v'})$ need to be maintain to cover (a, d).

In details, in encoding (a, d) for all alternative paths from a to d, a set of nodes W is used such that the removal of W disconnect all paths from a to d. It constructs 2-hop clusters based on $w \in W$ and any nodes that connect via

w are included in A_w and D_w. And all $w \in W$ are added into $L_{out}(a)$ and $L_{in}(d)$. Upon the deletion of a node w, it can safely remove w from all $L_{out}(a)$ and $L_{in}(d)$. It is because that if there is another path from a to d , there must be another $w' \in W$ such that $L_{out}(a)$ and $L_{in}(d)$ both contain w'. Note that the 2-hop cover compression ratio is in a relatively low priority in this regard.

8. 3-Hop Cover

Jin et al. in [25] propose a 3-Hop approach. Consider a transitive closure matrix for a DAG G (Figure 6.10). Suppose there exists a chain cover of G with k chains. Jin et al. show that the transitive closure matrix for G is a matrix of $k \times k$ blocks where each block is a Pseudo-upper triangular matrix. It can be done by ordering the nodes using their chain identifiers and then their positions in the chains. Jin et al. use $Con(G)$ to denote the set of pseudo-diagonal cells for all the blocks in the transitive closure matrix (the circled cells shown in Figure 6.10). It is easy to see that $Con(G)$ is enough to derive the transitive closure. $Con(G)$ can be easily calculated using Algorithm 2.

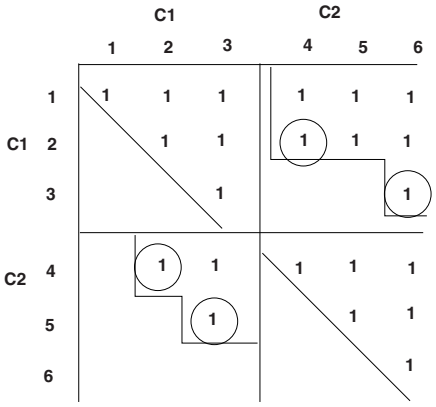


Figure 6.10. Transitive Closure Matrix

$Con(G)$ is already enough to answer a reachability query. But, the cost is high, because the number of nodes in $Con(G)$ can be large. Jin et al. encode $Con(G)$ using 3-hop cover codes. It is similar to the 2-hop cover codes. For every node u, there is a list of "entry points" $L_{in}(u)$ and a list of "exit points" $L_{out}(u)$. The difference between 2-hop and 3-hop is as follows. In a 2-hop cover code, u can reach v if any only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. But in a 3-hop cover code, it allows a point in $L_{out}(u)$ reach another point in $L_{in}(v)$ via a chain. Suppose that there is a chain $\cdots \rightsquigarrow v_i \rightsquigarrow \cdots \rightsquigarrow v_j \rightsquigarrow \cdots$. Then, $u \rightsquigarrow v$ is true if u can reach v_i (1st hop), v_i can reach v_j (2nd hop), and v_j can reach v (3rd hop). The algorithm to compute the 3-hop cover codes is similar to the algorithm to compute the 2-hop cover codes. The only difference

is that it needs to consider the set of pairs that can be encoded by each chain rather than each node. The time complexity for the 3-hop cover construction is $O(k \cdot n^2 \cdot |Con(G)|)$.

Given a 3-hop cover coding scheme encoding for $Con(G)$, it can answer a reachability query $u \rightsquigarrow v$ as follows: In the first step, it collects a set of entry points $L_{out}(u)$ can reach on the intermediate chains. In the second step, it collects a set of exit points which can reach v on the intermediate chains. Finally, it checks whether an entry point can reach an exit point using the chain ids and positions for nodes in the chain. The time complexity is $O(\log n + k)$ where n is the number of nodes in the graph G and k is the number of chains.

9. Distance-Aware 2-Hop Cover

The 2-hop cover coding schema discussed in the previous section can be used to answer reachability queries, $u \rightsquigarrow v$, but cannot be used to answer distance queries, $u \overset{\delta}{\rightsquigarrow} v$. A distance query $u \overset{\delta}{\rightsquigarrow} v$ is a reachability query $u \rightsquigarrow v$ with the shortest distance δ. In other words, it queries the shortest distance from u to v if it is reachable. Cohen et al. in [17] address this problem.

Consider an edge-weighted directed graph $G(E, V)$, where $\omega(u, v)$ represents the distance over the edge $(u, v) \in E$. Let $\delta(u, v)$ be the shortest distance from a node u to a node v. A 2-hop cover code of u is a pair of $L_{in}(u)$ and $L_{out}(u)$. Here, $L_{in}(u)$ is a set of pairs $\{(u_1, \delta(u_1, u)), (u_2, \delta(u_2, u)), \cdots\}$, and $L_{out}(u)$ is a set of pairs $\{(v_1, \delta(u, v_1)), (v_2, \delta(u, v_2)), \cdots\}$. A distance query $u \overset{\delta}{\rightsquigarrow} v$ is answered as

$$\min\{\delta(u, w) + \delta(w, v) | (w, \delta(u, w)) \in L_{out}(u) \wedge (w, \delta(w, v)) \in L_{in}(v)\}$$

It is worth nothing that the distance-aware 2-hop cover needs to maintain the additional shortest distance information.

Schenkel et al. in [30] discuss the distance-aware 2-hop cover. The algorithms in [30] can be used to compute the distance-aware 2-hop cover. However, in addition to the bottleneck in the third step, it needs high overhead to compute the shortest paths, and the resulting 2-hop cover can be unnecessarily large. Consider Figure 6.11. There is a subgraph G_i in which the node a is an ancestor of the nodes $x_1, x_2, \cdots, x_d$ in the subgraph G_i that appear in the cross-partition edges. As a result, all nodes, $x_1, x_2, \cdots, x_d$, appear in the skeleton graph. Assume that there is a 2-hop cluster, $S(A_w, w, D_w)$, in the skeleton graph, that contains all $x_1, x_2, \cdots, x_d$ in A_w. In computing the distance-aware 2-hop cover for G by augmenting the distance-aware 2-hop cover computed for the skeleton graph, it needs to identify the shortest path from a to w (Figure 6.11). There may exist many unnecessary pairs in the resulting distance-aware 2-hop cover such that $\delta(a, x) + \delta(x, w) > \delta(a, w)$.

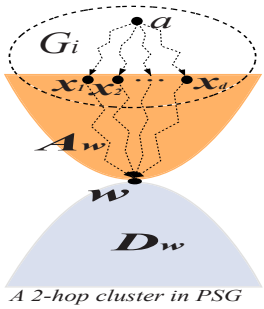


Figure 6.11. The 2-hop Distance Aware Cover (Figure 2 in [10])

Cheng and Yu in [10] discuss a new DAG-based approach and focus on two main issues.

- Issue-1: It cannot obtain a DAG G' for a directed graph G first, and compute the distance-aware 2-hop cover for G based on the distance-aware 2-hop cover computed for G'. In other words, it cannot represent a strongly connected component (SCC) in G as representative node in G'. It is because that a node w in a SCC on the shortest path from u to v does not necessarily mean that every node in the SCC is on the shortest path from u to v.
- Issue-2: The cost of dynamically selecting the best 2-hop cluster, in an iteration of the 2-hop cover program, cannot be reduced using the tree cover codes and R-tree as discussed in [13], because such techniques cannot handle distance information.

Cheng and Yu observe that if a 2-hop cluster, $S(A_w, w, D_w)$, is computed to cover all shortest paths containing the center node w, it can remove w from the underneath graph G, because there is no need to consider again any shortest paths via w any more.

Based on the observation, to deal with Issue-1, Cheng and Yu in [10] collapse every SCC into DAG by removing a small number of nodes from the SCC repeatedly until it obtains a DAG graph. To deal with Issue-2, when constructing 2-hop clusters, Cheng and Yu propose a new technique to reduce the 2-hop clusters by taking the already identified 2-hop clusters into consideration, to avoid storing unnecessary all-pairs of shortest paths.

Cheng and Yu propose a two-step solution. In the first phase, it attempts to obtain a DAG $G_{\downarrow}$ for a given graph G by removing a small number of nodes, $\hat{V}_{C_i}$, from every SCC, $C_i(V_{C_i}, E_{C_i})$. In computing a SCC $C_i(V_{C_i}, E_{C_i})$, every node, $w \in \hat{V}_{C_i}$ is taken as a center, and $S(A_w, w, D_w)$ is computed to cover shortest paths for the graph G. Then, all nodes in $\hat{V}_{C_i}$ will be removed, and

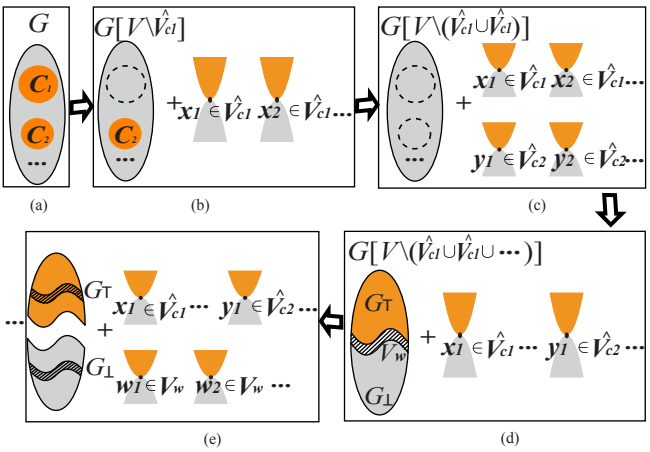


Figure 6.12. The Algorithm Steps (Figure 3 in [10])

a modified graph is constructed as an induced subgraph of $G(V, E)$, denoted as $G[V \setminus \hat{V}_{C_i}]$, with the set of nodes $V \setminus \hat{V}_{C_i}$. Figure 6.12(a) shows a graph G with several SCCs. Figure 6.12(b)-(d) illustrate the main idea of collapsing SCCs while computing 2-hop clusters. At the end, the original directed graph G is represented as a DAG G' plus a set of 2-hop clusters, $S(A_w, w, D_w)$, computed for every node, $w \in \hat{V}_{C_i}$. All shortest paths covered are the union of the shortest paths covered by all 2-hop clusters, $S(A_w, w, D_w)$, for every node, $w \in \hat{V}_{C_i}$, and the modified DAG G'. In the second phase, for the obtained DAG $G_{\downarrow}$, Cheng and Yu take the top-down partitioning approach to partition the DAG $G_{\downarrow}$, based on the early work in [14]. Figure 6.12(d)-(e) show that the graph can be partitioned hierarchically.

10. Graph Pattern Matching

In this section, we discuss several approaches to find graph patterns in a large data graph. A data graph is a directed node-labeled graph $G_D = (V, E, \Sigma, \phi)$. Here, V is a set of nodes, E is a set of edges (ordered pairs), Σ is a set of node labels, and ϕ is a mapping function which assigns each node, $v_i \in V$, a label $l_j \in \Sigma$. Below, we use $\mathsf{label}(v_i)$ to denote the label of node v_i. Given a label $l \in \Sigma$, the extent of l, denoted $\mathsf{ext}(l)$, is a set of nodes in G_D whose label is l. A graph pattern is a connected directed labeled graph $G_q = (V_q, E_q)$, where V_q is a subset of labels (Σ), and E_q is a set of edges (ordered pairs) between two nodes in V_q. There are two types of edges. Let $A, D \in V_q$. An edge $(A, D) \in E(G_q)$ represents a parent/child condition, denoted as $A \mapsto D$, which identifies all pairs of nodes, v_i and v_j, such that $(v_i, v_j) \in G_D$, $\mathsf{label}(v_i) = A$, and $\mathsf{label}(v_j) = D$. An edge $(A, D) \in E(G_q)$

represents a reachability condition, denoted as $A \hookrightarrow D$, that identifies all pairs of nodes, v_i and v_j, such that $v_i \rightsquigarrow v_j$ is true in G_D, for $\mathsf{label}(v_i) = A$, and $\mathsf{label}(v_j) = D$. A match in G_D matches the graph pattern G_q if it satisfies all the parent/child and reachability conditions conjunctively specified in G_q. A graph pattern matching query is to find all matches for a query graph. In this article, we focus on the reachability conditions, $A \hookrightarrow D$, and omit the discussions on parent/child conditions, $A \mapsto D$. We assume that a query graph G_p only consists of reachability conditions.

10.1 A Special Case: $A \hookrightarrow D$

In this section, we introduce three approaches to process $A \hookrightarrow D$ over a graph G_D.

Sort-Merge Join. Wang et al. propose a sort-merge join algorithm in [36] to process $A \hookrightarrow D$ over a directed graph using the tree cover codes [1]. Recall that for a given node u, $\mathsf{tccode}(u) = \{[u_{start_1}, u_{end_1}], [u_{start_2}, u_{end_2}], \cdots\}$, where u_{end_1} is the postorder when it traverses the spanning tree. We use $post(u)$ to denote the postorder of node u.

Let $Alist$ and $Dlist$ be two lists of $\mathsf{ext}(A)$ and $\mathsf{ext}(D)$, respectively. In $Alist$, every node v_i keeps all its intervals in the $\mathsf{tccode}(v_i)$. In $Dlist$, every node v_j keeps its unique postorder $post(v)$. Also, $Alist$ is sorted on the intervals $[s, e]$ by the ascending order of s and then the descending order of e, and $Dlist$ is sorted by the postorder number in ascending order. The sort-merge join algorithm evaluates $A \hookrightarrow D$ over G_D by a single scan on $Alist$ and $Dlist$ using the predicate $\mathcal{P}_{tc}(,)$. Wang et al. [36] propose a naive GMJ algorithm and an IGMJ algorithm which uses a range search tree to improve the performance of the GMJ algorithm.

Hash Join. Wang et al. also propose a hash join algorithm in [35] to process $A \hookrightarrow D$ over a directed graph using the tree cover codes. Unlike the sort-merge join algorithm, $Alist$ is a list of pairs $(val(u), post(u))$ for all $u \in ext(A)$. Here, $post(u)$ is the unique postorder of u, and $val(u)$ is either a start or an end of the intervals. Consider the node d in Figure 6.3(b), $post(d) = 7$, and there are two intervals, $[6, 7]$ and $[1, 4]$. In $Alist$, it keeps four pairs: $(6, 7)$, $(7, 7)$, $(1, 7)$, and $(4, 7)$. Like the sort-merge join algorithm, $Dlist$ keeps a list of postorders $post(v)$ for all $v \in \mathsf{ext}(D)$. $Alist$ is sorted in ascending order of $val(a)$ values, and $Dlist$ is sorted in ascending order of $post(d)$ values. The Hash Join algorithm, called HGJoin, is outline in Algorithm 5.

Join Index. Cheng et al. in [15] study a join index approach to process $A \hookrightarrow D$ using a join index built on top of G_D. The join index is built based on the 2-hop cover codes. We explain it using the same example given in [15].

Algorithm 5 HGJoin($Alist$, $Dlist$)

```
1:  H ← ∅;
2:  Output ← ∅;
3:  a ← Alist.first;
4:  d ← Dlist.first;
5:  while a ≠ Alist.last ∧ d ≠ Dlist.last do
6:     if val(a) ≤ post(d) then
7:        if post(a) ∉ H then
8:           hash post(a) into H;
9:           a ← a.next;
10:       else if val(a) < post(d) then
11:          delete post(a) from H;
12:          a ← a.next;
13:       else
14:          for all post(a) in H do
15:             append (post(a), post(d)) to Output;
16:          end for
17:          d ← d.next;
18:       end if
19:    else
20:       for all post(a) in H do
21:          append (post(a), post(d)) to Output;
22:       end for
23:       d ← d.next;
24:    end if
25: end while
26: return Output;
```

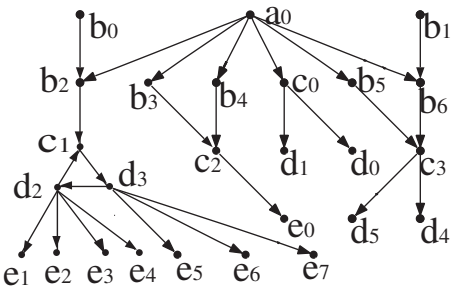


Figure 6.13. Data Graph (Figure 1(a) in [12])

A	A_{in}	A_{out}
a_0	$\emptyset$	$\{c_1, c_3\}$

B	B_{in}	B_{out}
b_0	$\emptyset$	$\{c_1\}$
b_1	$\emptyset$	$\{c_3, b_6\}$
b_2	$\{a_0, b_0\}$	$\{c_1\}$
b_3	$\{a_0\}$	$\{c_2\}$
b_4	$\{a_0\}$	$\{c_2\}$
b_5	$\{a_0\}$	$\{c_3\}$
b_6	$\{a_0\}$	$\{c_3\}$

C	C_{in}	C_{out}
c_0	$\{a_0\}$	$\emptyset$
c_1	$\emptyset$	$\emptyset$
c_2	$\{a_0\}$	$\emptyset$
c_3	$\emptyset$	$\emptyset$

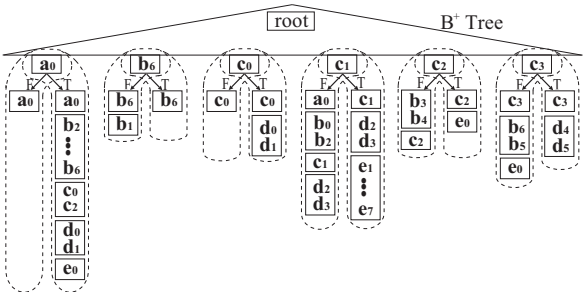
D	D_{in}	D_{out}
d_0	$\{a_0, c_0\}$	$\emptyset$
d_1	$\{a_0, c_0\}$	$\emptyset$
d_2	$\{c_1\}$	$\{c_1\}$
d_3	$\{c_1\}$	$\{c_1\}$
d_4	$\{c_3\}$	$\emptyset$
d_5	$\{c_3\}$	$\emptyset$

E	E_{in}	E_{out}
e_0	$\{a_0, c_2\}$	$\emptyset$
e_1	$\{c_1\}$	$\emptyset$
$\vdots$	$\vdots$	$\vdots$
e_7	$\{c_1\}$	$\emptyset$

(a) Five Lists

(A,B)	$\{a_0\}$	(A,C)	$\{a_0, c_1, c_3\}$	(D,E)	$\{c_1\}$
(A,E)	$\{a_0, c_1\}$	(B,C)	$\{c_1, c_2, c_3\}$	(C,E)	$\{c_1, c_2\}$
(B,E)	$\{c_1, c_2\}$	(C,D)	$\{c_0, c_1, c_3\}$	(D,C)	$\{c_1\}$
(B,D)	$\{c_1, c_3\}$	(A,D)	$\{a_0, c_1, c_3\}$	(D,D)	$\{c_1\}$
(B,B)	$\{b_0, b_6\}$	(C,C)	$\{c_0, c_1, c_2, c_3\}$		

(b) W-table



(c) A Cluster-Based R-Join-Index

Figure 6.14. A Graph Database for G_D (Figure 2 in [12])

Consider a graph G_D (Figure 6.13). The 2-hop cover codes for all nodes in G_D are shown in Figure 6.14(a). It is a compressed 2-hop cover code which removes $v \rightsquigarrow v$ from the 2-hop cover code computed. The predicate $\mathcal{P}_{2hop}(,)$ is slightly modified using the compressed 2-hop cover codes as follows.

$$\mathcal{P}_{2hop}(\mathsf{2hopcode}(u), \mathsf{2hopcode}(v)) = L_{out}(u) \cap L_{in}(v) \neq \emptyset \vee u \in L_{in}(v) \vee v \in L_{out}(u)$$

A cluster-based join index for a data graph G_D based on the 2-hop cover computed, $\mathcal{H} = \{S_{w_1}, S_{w_2}, \cdots\}$, where $S_{w_i} = S(A_{w_i}, w_i, D_{w_i})$ and all w_i are centers. It is a B$^+$-tree in which its non-leaf blocks are used for finding a given center w_i. In the leaf nodes, for each center w_i, its A_{w_i} and D_{w_i}, denoted *F-cluster* and *T-cluster*, are maintained. A w_i's *F-cluster* and *T-cluster* are further divided into labeled *F-subclusters*/*T-subclusters* where every node, a_i, in an A-labeled *F-subcluster* can reach every node d_j in a D-labeled *T-subcluster*, via w_i. Together with the cluster-based join index, it designs a W-table in which, an entry $W(X, Y)$ is a set of centers. A center w_i will be included in $W(A, B)$, if w_i has a non-empty A-labeled *F-subcluster* and a non-empty D-labeled *T-subcluster*. It helps to find the centers, w_i, in the cluster-based join index, that have an A-labeled *F-subcluster* and a D-labeled *T-subcluster*. For the cluster-based join index for G_D (Figure 6.13) is shown in Figure 6.14(c), and the W-table is shown in Figure 6.14(b). Consider $A \hookrightarrow B$. The entry $W(A, B)$ keeps $\{a_0\}$, which suggests that the answers can be only found in the clusters at the center a_0. As shown in Figure 6.14(c), the center a_0 has an A-labeled *F-subcluster* $\{a_0\}$, and a B-labeled *T-subcluster* $\{b_2, b_3, b_4, b_5, b_6\}$. The answer is the Cartesian product between these two labeled subclusters. It can process $A \hookrightarrow D$ queries efficiently.

Cheng et al in. [11] discuss performance issues between the sort-merge join approach and the index approach.

10.2 The General Cases

Chen et al. in [8] propose a holistic based approach for graph pattern matching. But, a query graph, G_q, is restricted to be a tree, which we introduce in brief in Section 2. Their *TwigStackD* algorithm process a tree-shaped G_q in two steps. In the first step, it uses *Twig-Join* algorithm in [7] to find all patterns in the spanning tree of G_D. In the second step, for each node popped out from the stacks used in *Twig-Join* algorithm, *TwigStackD* buffers all nodes which at least match a reachability condition in a bottom-up fashion, and maintains all the corresponding links among those nodes. When a top-most node that matches a reachability condition, *TwigStackD* enumerates the buffer pool and outputs all fully matched patterns. *TwigStackD* performs well for very sparse data graphs. But, its performance degrades noticeably when the G_D becomes dense, due to the high overhead of accessing edge transitive closures.

Cheng et al. in [11, 12] consider $A \hookrightarrow D$ as a *R*-join (like θ-join), and process a graph pattern matching as a sequence of *R*-joins. The issue is how to select join order. They propose a dynamic programming algorithm to determine the *R*-join order in [11]. They also propose an *R*-join/*R*-semijoin approach in [12]. The basic idea is to divide the join-index based approach into two steps namely filter and fetch. The filter steps shares the similarity with semijoin, and the fetch step is to join. Cheng et al. study how to select *R*-join/*R*-semijoin order by interleaving *R*-joins with *R*-semijoins, using dynamic programming in [12].

Wang et al. in [35] propose a query graph G_q based on the hash join approach, and consider how to share the processing cost when it needs to process several $Alist$ and $Dlist$ simultaneously. Wang et al. propose three basic join operators, namely, IT-HGJoin, T-HGJoin, and Bi-HGJoin. The IT-HGJoin processes a subgraph of a query with one descendant and multiple ancestors, for example, $A \hookrightarrow D \wedge B \hookrightarrow D$. The T-HGJoin process a subgraph of a query with one ancestor and multiple descendants, for example, $A \hookrightarrow C \wedge A \hookrightarrow D$. The Bi-HGJoin processes a complete bipartite subgraph of a query with multiple ancestors and multiple descendants, for example $A \hookrightarrow C \wedge A \hookrightarrow D \wedge B \hookrightarrow C \wedge B \hookrightarrow D$. A general query graph G_q will be processed by a set of subgraph queries using IT-HGJoin, T-HGJoin, and Bi-HGJoin.

11. Conclusions and Summary

In this chapter, we presented a survey on reachability queries. We discussed several coding-based approaches using traversal, dual-labeling, tree cover, chain cover, path-tree cover, 2-hop cover, and 3-hop cover approaches. We also addressed how to support distance-aware queries such as to find the shortest distance between two nodes in a large directed graph using the 2-hop cover, and how to support graph pattern matching using the existing graph-based coding schema. As future work, it becomes important how to use the graph-based coding schema to support more real large graph-based applications.

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 (SIGMOD 1989)*, 1989.

[2] K. Anyanwu and A. Sheth. ρ-queries: enabling querying for semantic associations on the semantic web. In *Proceedings of the 12th international conference on World Wide Web (WWW 2003)*, 2003.

[3] B. Berendt and M. Spiliopoulou. Analysis of navigation behaviour in web sites integrating multiple information systems. *The VLDB Journal*, 9(1), 2000.

[4] R. Bramandia, J. Cheng, B. Choi, and J. X. Yu. Updating recursive XML views without transitive closure. To appear in *VLDB J.*, 2009.

[5] R. Bramandia, B. Choi, and W. K. Ng. On incremental maintenance of 2-hop labeling of graphs. In *Proceedings of the 17th international conference on World Wide Web (WWW 2008)*, 2008.

[6] D. Brickley and R. V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. W3C Recommendation, 2000.

[7] 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 (SIGMOD 2002)*, 2002.

[8] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proceedings of the 31nd international conference on Very large data bases (VLDB 2005)*, 2005.

[9] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *Proceedings of the 24th International Conference on Data Engineering (ICDE 2008)*, 2008.

[10] J. Cheng and J. X. Yu. On-line exact shortest distance query processing. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT 2009)*, 2009.

[11] J. Cheng, J. X. Yu, and B. Ding. Cost-based query optimization for multi reachability joins. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA 2007)*, 2007.

[12] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *Proceedings of the 24th International Conference on Data Engineering (ICDE 2008)*.

[13] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT 2006)*, 2006.

[14] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *Proceedings of the 11th International Conference on Extending Database Technology (EDBT 2008)*, 2008.

[15] J. Cheng, J. X. Yu, and N. Tang. Fast reachability query processing. In *Proceedings of the 11th International Conference on Database Systems for Advanced Applications (DASFAA 2006)*, 2006.

[16] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

[17] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the 13th annual ACM-SIAM symposium on Discrete algorithms (SODA 2002)*, 2002.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2001.

[19] S. DeRose, E. Maler, and D. Orchard. XML linking language (XLink) version 1.0. 2001.

[20] S. DeRose, E. Maler, and D. Orchard. XML pointer language (XPointer) version 1.0. 2001.

[21] J. Edmonds. Optimum branchings. *J. Research of the National Bureau of Standards*, 71B:233–240, 1967.

[22] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Rec.*, 26(3), 1997.

[23] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management (CIKM 2005)*, pages 594–601, 2005.

[24] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[25] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-HOP: A high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD international conference on Management of data (SIGMOD 2009)*, 2009.

[26] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD 2008)*, 2008.

[27] D. S. Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the 5th annual ACM symposium on Theory of computing (STOC 1973)*, 1973.

[28] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the 36 annual ACM symposium on Theory of computing (STOC 2004)*, 2004.

[29] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex XML document collections. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT 2004)*, 2004.

[30] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the HOPI index for complex XML document collections. In *Proceedings of the 21th International Conference on Data Engineering (ICDE 2005)*, 2005.

[31] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58(1-3):325–346, 1988.

[32] S. TrißI and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD 2007)*, 2007.

[33] J. van Helden, A. Naim, R. Mancuso, , M. Eldridge, L. Wernisch, D. Gilbert, and S. Wodak. Reresenting and analysing molecular and cellular function using the computer. *Journal of Biological Chemistry*, 381(9-10), 2000.

[34] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of the 22th International Conference on Data Engineering (ICDE 2006)*, 2006.

[35] H. Wang, J. Li, J. Luo, and H. Gao. Hash-base subgraph query processing method for graph-structured XML documents. *Proceedings VLDB Endowment*, 1(1), 2008.

[36] H. Wang, W. Wang, X. Lin, and J. Li. Labeling scheme and structural joins for graph-structured XML data. In *Proceedings of the 7th Asia-Pacific Web Conference on Web Technologies Research and Development (APWeb 2005)*, 2005.