3-HOP: A High-Compression Indexing Scheme for Reachability Query

Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry
Department of Computer Science, Kent State University
Kent, OH 44242, USA
{jin,yxiang,nruan,dfuhry}@cs.kent.edu

ABSTRACT

Reachability queries on large directed graphs have attracted much attention recently. The existing work either uses spanning structures, such as chains or trees, to compress the complete transitive closure, or utilizes the 2-hop strategy to describe the reachability. Almost all of these approaches work well for very sparse graphs. However, the challenging problem is that as the ratio of the number of edges to the number of vertices increases, the size of the compressed transitive closure grows very large. In this paper, we propose a new 3-hop indexing scheme for directed graphs with higher density. The basic idea of 3-hop indexing is to use chain structures in combination with hops to minimize the number of structures that must be indexed. Technically, our goal is to find a 3-hop scheme over dense DAGs (directed acyclic graphs) with minimum index size. We develop an efficient algorithm to discover a *transitive closure contour*, which yields near optimal index size. Empirical studies show that our 3-hop scheme has much smaller index size than state-of-the-art reachability query schemes such as 2-hop and path-tree when DAGs are not very sparse, while our query time is close to path-tree, which is considered to be one of the best reachability query schemes.

Categories and Subject Descriptors

H.2.8 [**Database management**]: Database Applications—*graph indexing and querying*

General Terms

Performance

Keywords

Graph indexing, Reachability queries, Transitive closure, 3-Hop, 2-Hop, Path-tree

1. INTRODUCTION

The rapid accumulation of very large graphs from a diversity of disciplines, such as biological networks, social networks, ontologies, XML, and RDF databases, among others, calls for the graph database system. Important research issues, ranging theoretical foundations including algebra and query language [2], to indices for various graph queries [20, 12] and more recently, graph OLAP/summarization [17], have attracted much recent attention. Among them, graph reachability query processing has evolved into a core problem: *given two vertices u and v in a directed graph, is there a path from u to v ($u \to v$)*?

Graph reachability is one of the fundamental research questions across several disciplines in computer science, such as software engineering and distributed computing. In the database research community, the initial interest in reachability queries has been driven by the need to handle recursive queries, with focus on efficient and effective transitive closure compression. Recently, this problem has captured the attention of database researchers again, due to the increasing importance of XML data management, and fast growing graph data, such as large scale social networks, WWW, and biological networks. For instance, in XML databases, the reachability query is the basic building block for the typical path query format $//P_1//P_2// \cdots //P_m$, where "//" is the ancestor-descendant search and P_i is the tag. Reachability queries also have an important role for managing/querying RDF and domain ontologies. In bioinformatics, reachability queries can be used to answer basic gene regulation questions in the regulatory network.

1.1 Prior Work

In order to tell whether a vertex u can reach another vertex v in a directed graph, many approaches have been developed over the years. For a reachability query, we can effectively transform a directed graph into a directed acyclic graph (DAG) by coalescing strongly connected components into vertices and utilizing the DAG to answer the reachability queries. Thus, throughout the paper, we will only focus on DAG. Let $G = (V, E)$ be the DAG for a reachability query. In Table 1.1, we summarize these approaches in terms of their index size, construction time, and query processing time based on worst-case analysis. Here, n is the number of vertices ($n = |V|$) and m is the number of edges ($m = |E|$). Parameter k is the width of the chain decomposition of DAG G [11], t is the number of (non-tree) edges left after removing all the edges of a spanning tree of G [19], and k' is the width of the path decomposition [12]. These three parameters k, t and k', are method-specific and will be explained in more detail when we discuss their corresponding methods.

DFS/BFS and Transitive Closure Computation: We first discuss two classical approaches for reachability query, representing two extremes with regard to index size and query time. DFS/BFS needs to traverse the graph online and can take up to $O(n + m)$ time to answer a reachability query. This is too slow for large graphs. The second approach precomputes the transitive closure of G, i.e., it records the reachability between every pair of vertices in advance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.
Copyright 2009 ACM 978-1-60558-551-2/09/06 ...$5.00.

	Index Size	Construction Time	Query Time
DFS/BFS	$O(n+m)$	$-$	$O(n+m)$
Transitive Closure [16]	$O(n^2)$	$O(nm)$	$O(1)$
Opt. Chain Cover [11]	$O(nk)$	$O(n^3)$	$O(\log k)$
Opt. Chain Cover [5]	$O(nk)$	$O(n^2+kn\sqrt{k})$	$O(\log k)$
Opt. Tree Cover [1]	$O(n^2)$	$O(nm)$	$O(\log n)$
Dual Labeling [19]	$O(n+t^2)$	$O(n+m+t^3)$	$O(1)$
Labeling+SSPI [4]	$O(n+m)$	$O(n+m)$	$O(m-n)$
GRIPP [18]	$O(m+n)$	$O(n+m)$	$O(m-n)$
Path-Tree [12]	$O(nk')$	$O(mk')$/$O(mn)$	$\log^2 k'$
2-Hop [9]	$\tilde{O}(nm^{1/2})$	$O(n^3\lvert T_c\rvert)$	$\tilde{O}(m^{1/2})$

Table 1: Worst-Case Complexity

While this approach can answer reachability queries in constant time, its storage cost $O(n^2)$ is prohibitive for large graphs.

Indeed, tackling the storage cost by effectively compressing the transitive closure has been the major theme of index construction for graph reachability processing. Typically, however, improved compression comes at the cost of slower query answering time. To find the right balance between transitive closure compression and reasonable query answering time is the driving force of ongoing research into graph reachability indexing.

The existing research largely falls into two categories: the first category attempts to apply simple graph structures, such as chains and trees, to compress the transitive closure of a DAG. The optimal chain cover, tree cover and the recent path-tree cover all belong to this category. The second category, referred to as 2-hop indexing, tries to encode the reachability using a subset of vertices which serve as intermediaries, i.e., each vertex records a list of intermediate vertices it can reach and a list of intermediate vertices which can reach it. Then, 2-hop reachability means the starting vertex can reach an intermediate vertex (the first hop) and this intermediate vertex can reach the end vertex (the second hop). In the following, we go through these approaches in more detail.

Optimal Chain Cover: The basic idea of optimal chain cover is to decompose a DAG into a minimal number of pair-wise disjoint chains, and then assign each vertex in the graph a chain ID and its sequence number in its chain. Given this, if a vertex can reach another chain, it records only the smallest vertex it reaches in that chain. In other words, each vertex in the compressed transitive closure covers the remaining vertices (all the vertices with a higher sequence number) in its respective chain. To determine if vertex u reaches vertex v, we only need to check if u reaches any vertex (say, v') in v's chain, and if yes, we check if the vertex v' has a smaller sequence number than v. This strategy can compress the transitive closure since we need to record at most one vertex in each chain for a given vertex. If the minimal number of chains for a DAG (also referred to as the *width* of the DAG) is k, then this approach has $O(nk)$ index size and $O(\log k)$ query time.

Jagadish [11] pioneered the application of chain decomposition in the database research community to compress the transitive closure. He demonstrated that the problem of finding the minimal number of chains from G can be transformed into a network flow problem, which can be solved in $O(n^3)$. He also proposed several heuristic algorithms for chain decomposition in order to reduce the computational cost and actual index size. Recently, Cheng [7] proposed an $O(n^2+kn\sqrt{k})$ time algorithm to decompose a DAG into a minimal number of chains.

The worst case complexity of the chain cover approach is clearly decided by the width of DAG. If the width is high, we tend to have a lot of chains with only a small number of vertices, resulting in a high index cost. Another way to look at the compression rate is by observing that each vertex in compressed transitive closure covers a partial chain (from the vertex itself to the last vertex in the chain). Let $R(u)$ be the transitive closure of u. Let $R^C(u)$ be the number of vertices u records for the chain decomposition. Then, the compression ratio of the chain decomposition is defined as $\frac{\sum_{u \in V}|R(u)|}{\sum_{u \in V}|R^C(u)|}$. Thus, we can see that the compression ratio is exactly the average size of the partial chains each vertex in the compressed transitive closure covers.

Optimal Tree Cover and Its Variants: The optimal tree cover utilizes a (spanning) tree to compress the transitive closure [1]. Each vertex in the tree is labeled by a pair of numbers, corresponding to an interval: if a vertex is an ancestor of another vertex in the tree, the interval labeling guarantees that the interval of the first vertex contains the interval of the second vertex. Note that if a vertex reaches the root of a subtree in the original DAG, it will reach all the vertices in the subtree. Thus, for each vertex in the DAG, we can organize all the vertices in its transitive closure, i.e., all the vertices it can reach, into pair-wise disjoint subtrees. To compress the transitive closure, for each subtree, we only need to record its root vertex. To answer the reachability query from vertex u to vertex v, we check if the interval of v is contained by any interval associated with those subtree roots we have recorded for u.

Agrawal *et al.* [1] formally introduced the tree cover and found an optimized algorithm to discover a tree cover which can maximally compress the transitive closure. They also showed that the tree cover approach can provide a better compression rate than the optimal chain cover approach. The advantage of the tree cover approach over the chain cover approach comes from the fact that each tree-cover vertex covers an entire subtree, while each chain-cover vertex covers only a partial chain.

Several recent studies focus on the tree cover approach and try to improve either its query processing time and/or provide a smaller index size. Wang *et al.* [19] develop the Dual-Labeling approach which tries to improve the query time and index size for very sparse graphs, where the number of non-tree edges t is much smaller than the number of vertices n ($t << n$). Their approach can reduce the index size to $O(n+t^2)$ and achieve constant query answering time. Unfortunately, many real world graphs would not satisfy the condition required by this approach, and when $t > n$, this approach will not help compress the index size.

Label+SSPI [4] and GRIPP [18] aim to minimize the index construction time and index size. They achieve $O(m+n)$ index construction time and $O(m+n)$ index size. However, this is at the sacrifice of the query time, which will cost $O(m-n)$. Both algorithms start by extracting a tree cover and then deploy an online search algorithm utilizing the tree structure to speed up the DFS process.

Path-Tree Cover: The latest work to use a simple graph structure to compress transitive closure is the path-tree cover approach, proposed by Jin *et al.* [12], which generalizes the tree cover approach. They observe that the covering capability of each vertex in the compressed transitive closure is determined by the number of parents and children each vertex has in the simple graph structure. For instance, a chain vertex has one parent and one child while a tree vertex has one parent and multiple children. The path-tree allows two parents and multiple children. In path-tree cover, all vertices in the original DAG are partitioned into pair-wise disjoint paths (k' is the number of paths in the path-decomposition for a DAG G), and then those paths serve as vertices in a tree structure. In other words, the path-tree utilizes a tree-like structure, where each vertex represents a path in the original DAG. Each vertex in the path-tree needs only three numbers, two numbers for the interval label of the tree-structure and one sequence number from a DFS traversal pro-

cedure, to answer the reachability query between any two vertices in the path-tree in constant time. In [12], authors proposed two path-tree schemes, PTree-1 and PTree-2. PTree-1 utilizes optimal tree cover and thus has $O(mn)$ construction time while PTree-2 has $O(mk)$ construction time.

Given this, to compress the transitive closure, a vertex u only needs to record vertex v, such that 1) $u \rightarrow v$ and 2) there is no vertex v' such that $u \rightarrow v'$ and v' can reach v in the path-tree. Theoretically, they prove that path-tree cover can always perform the compression of transitive closure better than or equal to the optimal tree cover approaches and chain cover approaches. Note that the enhanced power of the path-tree cover is a consequence of the increased parent/child connectivity of path-tree vertices vs. tree cover or chain cover vertices.

2-HOP Indexing: The 2-hop labeling method proposed by Cohen *et al.* [9] is quite different from the aforementioned simple graph covering approaches. It compresses the transitive closure using a subset of intermediate vertices. Each vertex records a list of intermediate vertices it can reach and a list of intermediate vertices which can reach it. The index size is the total number of intermediate vertices each vertex records. They propose an approximate (greedy) algorithm based on set-covering which can produce a 2-hop cover with size no larger than the minimum possible 2-hop indexing by a logarithmic factor. The minimum 2-hop index size is conjectured to be $\tilde{O}(nm^{1/2})$.

The major problem of the 2-hop indexing approach is its high construction cost. The greedy set-covering algorithm needs to iteratively find a subset of vertices which utilizes a candidate vertex as the intermediate hop. The subset of vertices are selected to minimize the *price* measure, i.e., the cost of recording such an intermediate hop of these vertices with respect to the number of uncovered reachable vertex pairs in this subset. Finding the subset of vertices with minimal price can be transformed into the problem of finding a densest subgraph in a bipartite graph. The approximate algorithm to solve this subproblem is in the linear order with respect to the number of edges in the bipartite graph. Besides, each vertex in the DAG can serve as the intermediate hop which corresponds to a bipartite graph. Thus, for each iteration, it takes $O(n^3)$ to find such a desired subset of vertices. Considering the iteration needs to cover the entire transitive closure T_c, we can see its construction time is $O(n^3|T_c|)$.

Several approaches have been proposed to reduce its construction time. Schenkel *et al.* propose the HOPI algorithm, which applies a divide-and-conquer strategy to compute 2-hop labeling [15]. Recently, Cheng *et al.* propose several methods, such as a geometric-based algorithm [6] and graph partition technique [7], to produce a 2-hop labeling. Though their algorithms significantly speed up the 2-hop construction time, they do not produce the approximation bound of the labeling size which is produced by Cohen *et al.*'s approach.

1.2 Our Contribution

Almost all these approaches work reasonably well for very sparse graphs (where the number of edges is very close to the number of vertices). However, as the ratio of the number of edges to the number of vertices increases, the size of the compressed transitive closure of the simple graph covering approaches can grow very large. In many real world graphs, such as citation networks, the semantic web, and biological networks, the number of edges can be several times the number of vertices. In general, the simple graph covering approach works well only for those DAGs which have a structure similar to the building-block chain, tree, or path-tree structures. However, in many real world graphs, since edge density is much higher than in simple graph structures, many edges will be left uncovered. Vertices of uncovered edges likely need to be recorded as ancillary data in the compressed transitive closure of the DAG, increasing the index size. Thus, the size of the compressed transitive closure can become very large as the density grows.

The original 2-hop [9] builds on top of the set-covering framework and is theoretically appealing as it achieves a guaranteed approximation bound. However, to our knowledge, there is little theoretical comparison between the 2-hop approach and the simple graph covering approaches in existing research. Most studies do not even empirically compare the 2-hop approach and the simple graph covering approaches. This may be due in part to the 2-hop approach not scaling well to large graphs, even graphs with only thousands of vertices. Specifically, since the original 2-hop needs to compute the complete transitive closure, it becomes very expensive as the edge density of the graph becomes larger. Though several heuristic techniques [15, 6, 7] have been proposed to construct 2-hop faster, they do not guarantee any approximation bound as the original 2-hop does. None of these methods have compared their compression ratio directly with the optimal 2-hop approaches, even on relatively small graphs.

To summarize, the major research challenge for existing graph reachability indexing is how to significantly compress the transitive closure when the ratio between the number of edges and the number of vertices increases. Driven by this need, we propose a new *3-hop* indexing scheme for directed graphs with higher density. The basic idea in 3-hop indexing is to utilize a simple graph structure, rather than a sole vertex, as an intermediate hop to describe the reachability between source vertices and destination vertices. In this paper, we focus on the chain structure. The new indexing scheme does not need to compute the entire transitive closure. Instead, it only needs to compute and record a number of so-called "contour" vertex pairs, which can be orders of magnitude smaller than the size of the transitive closure. Indeed, it is even much smaller than the compressed transitive closure of the chain cover. The connectivity of any pair of vertices in the DAG can be answered by those contour vertex pairs. Further, we "factorize" these contour vertex pairs by recording a list of "entry points" and "exit points" on some intermediate chains. We derive an efficient algorithm to generate an index which approximates the minimal 3-hop indexing by a logarithmic factor. Theoretically, we show that 3-hop labeling always has a better minimal compression ratio than 2-hop labeling, and its construction time is much faster than that of 2-hop.

We perform a detailed experimental evaluation on both real and synthetic datasets by comparing 3-hop labeling, 2-hop labeling and the state-of-the-art path-tree covering approach. Empirical studies show that our 3-hop scheme has a much smaller index size than prior state-of-art reachability query schemes for dense DAGs when the number of edges is not close to the number of vertices, i.e., $|E| \not\approx |V|$. The query processing time of 3-hop is close to path-tree's, which is considered to be one of the best reachability query schemes.

2. BASIC IDEAS OF 3-HOP INDEXING

2.1 Basic 3-Hop

The 3-hop reachability indexing is analogous to the highway system of the transportation network. To reach a destination from a starting point, you simply need to get on an appropriate highway and get off at the right exit to get to the destination. The highway system in the 3-hop labeling is simple graph structures, such as chains or trees, as they can encode the reachability information using a constant labeling size. In this paper, we focus on utilizing

chains, i.e., each chain serves as a different highway. Since each chain has a direction, each vertex u records a list of "entry points" (the smallest vertices) it can reach on some chains. It also records a list of "exit points" (the largest vertices) which can reach it in some chains. Here, the order of vertices in the chain is with respect to their topological order in that chain, i.e., a vertex with a smaller number can reach a vertex with a larger number.

Given this, the three hops are 1) the first hop from the starting vertex to the entry point of some chain, 2) the second hop from the entry point in the chain to the exit point of the chain, and finally 3) the third hop from the exit point of the chain to the destination vertex. The goal of 3-hop indexing is to assign all vertices with a minimal total number of entry and exit points so that they can maximally compress the transitive closure.

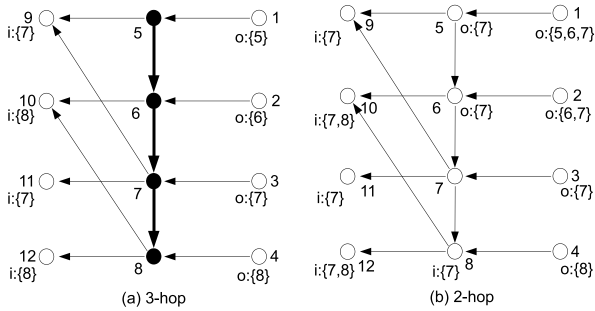

Figure 1: A simple example for 3-hop and 2-hop

Figure 1(a) shows an example using the chain $5 \rightarrow 6 \rightarrow 7 \rightarrow 8$ as the intermediate hop (or highway). Thus, each vertex not on the chain only needs to record its entry point and exit point in that chain, listing them in the set o and set i associated with each vertex, respectively. To tell if vertex 2 can reach 9, we compare 2's entry point with 9's exit point. We conclude that 2 can reach 9 because 2's entry point of 6 precedes exit point 7, which then reaches vertex 9. In total, the simple 3-hop scheme records 8 vertices to encode the transitive closure by using a single chain.

Figure 1(b), shows the optimal 2-hop labeling where each vertex records a list of intermediate vertices it reaches and a list of vertices which reach it. Here, 2-hop needs to record a total of 16 vertices to encode the transitive closure. However, readers should be advised that this is a very simple and incomplete example giving the basic idea of 3-hop. Detailed definitions, algorithms and complete running examples of 3-hop will be given from now on.

2.2 Chain Decomposition for 3-Hop

A simple technique which can significantly boost the 3-hop compression ratio is to apply a chain decomposition for the entire DAG first. For the 3-hop perspective, such a decomposition would associate each vertex itself with a highway since each vertex is partitioned to a chain. This suggests that many vertices in the same chain may share the same entry points and exit points of some other chains. Thus, we do not need to explicitly record those points for each of these vertices in the same chain, and therefore can further compress the transitive closure. To better understand the intuition of boosting 3-hop with a chain decomposition, let us see the running example in Figure 2.

Figure 2 is a DAG with 4 chains as a result of chain decomposition. In 3-hop, each chain serves as a highway and each vertex also belongs to a highway. In Figure 3, we show the vertices using

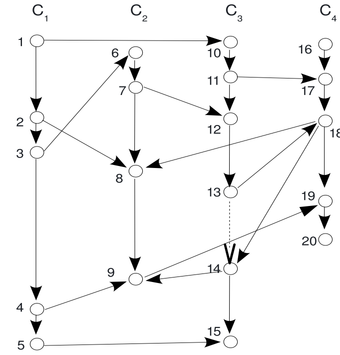

Figure 2: A simple DAG with a chain decomposition. (Dotted arrow from $13 \rightarrow 14$ is not an edge in the original DAG, but an inferred one using reachability).

chains C_2 and C_3 as intermediate hops (highways) to encode their transitive closure. At the left of each chain, we draw those vertices which record an entry point into that chain, and at the right of each chain, we draw those vertices which record an exit point out of the corresponding chain. To be more efficient, we organize into an "outgoing" segment those consecutive vertices (on one chain) which share the same entry point, and correspondingly we organize into an "incoming" segment those consecutive vertices (on one chain) which share the same exit point.

Specifically, we organize all the vertices on the left which share the same entry point into an "outgoing" segment, and all the vertices on the right which share the same exit point into an "incoming" segment. Each segment corresponds to a list of consecutive vertices in a chain. For instance, the vertices in the outgoing segment from 1 to 3 all record vertex 6 in chain C_2 as the entry point, and they are the first three vertices in chain C_1. The vertices in the incoming segment from 17 to 20 all record vertex 11 in chain C_3 as the exit point, and they are the last four vertices in chain C_4.

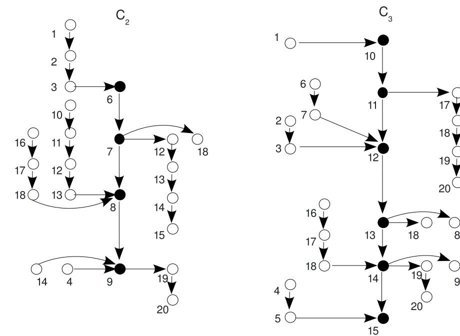

Figure 3: Two examples of Reachability between segments (through chains C_2 and C_3)

Intuitively, we can apply 3-hop with chain decomposition to answer a reachability query. For example, to answer whether vertex 6 can reach vertex 19, we find that vertex 6 is in segment $(6, 7)$ which can reach vertex 12 in C_3, and vertex 19 is in segment $(19, 20)$ which can be reached by vertex 14 in C_3. Then we say 6 can reach

19 because 6 can reach 12, 19 can be reached by 14, and 12 reaches 14 in the chain C_3.

2.3 3-Hop Indexing and Our Approach

The major research problem we will study in this paper is as follows: *Given a chain decomposition $\{C_1, C_2, \ldots, C_k\}$ of a DAG G, how can we utilize 3-hop strategy to maximally compress the transitive closure and answer reachability queries efficiently?* Our approach addresses this problem in three steps:

1.(Section 3) Given a chain decomposition, we first derive a concise representation of the transitive closure, called the *contour* of the transitive closure. This representation allows us to quickly identify those vertices which share the same entry point and vertices which share the same exit point.

2.(Section 4) We show that a 3-hop strategy which maximally compresses the contour corresponds to a generalized "factorization" of the contour. We develop an efficient greedy algorithm to approximate the optimal results within a logarithmic factor.

3.(Section 5) We provide a query processing procedure utilizing the index based on the 3-hop compression of the transitive closure contour. We also derive a theoretically faster query processing scheme by transforming 3-hop contour into a *3-hop segment* indexing.

3. TRANSITIVE CLOSURE CONTOUR

In this section, we will study a concise representation of the transitive closure matrix based on the chain decomposition of the DAG. This representation will form the basis for efficient construction of the 3-HOP index. We will derive a fast algorithm to directly generate this concise representation.

3.1 Notation and Chain-Decomposition

Let $G = (V, E)$ be a directed acyclic graph (DAG), where $V = \{1, 2, \cdots, n\}$ is the vertex set, and $E \subseteq V \times V$ is the edge set. We use (v, w) to denote the edge from vertex v to vertex w, and we use $(v_0, v_1, \cdots, v_p)$ to denote a *path* from vertex v_0 to vertex v_p, where (v_i, v_{i+1}) is an edge ($0 \le i \le p-1$). In a DAG, all paths are simple paths, meaning each vertex in a path is distinct. We say vertex v is *reachable* from vertex u (denoted as $u \to v$) if there is a path starting from u and ending at v.

A *chain* is the generalization of path, which is also a sequence of vertices, $(v_0, v_1, \cdots, v_p)$, where v_{i+1} is reachable from v_i ($v_i \to v_{i+1}, 0 \le i \le p-1$). Clearly, any path in G is also a chain. However, the reverse is not necessarily true (see chain C_3 in Figure 2). Let C_1 and C_2 be two chains of G. We use $C_1 \cap C_2$ to denote the set of vertices appearing in both chains and use $C_1 \cup C_2$ to denote the set of vertices appearing in either of the chains.

DEFINITION 1. **(Chain Decomposition)** *A* chain decomposition *of DAG $G = (V, E)$ is a collection of pair-wise distinct chains, $C_1, C_2, \cdots, C_k$, such that $C_1 \cup C_2 \cup \cdots C_k = V$ and $C_i \cap C_j = \emptyset$, for any $i \neq j$. The integer k is called the* width *of the decomposition.*

Given the chain decomposition, we assign to each vertex v a pair of IDs, (cid,oid), where cid is the ID of the chain vertex v belongs to, and oid is v's relative order on the chain. For any two vertices u and v in the same chain, we have $u \preceq v$ iff $u.oid \le v.oid$. If $u.oid < v.oid$, we also say u is smaller than v and vice versa. Several algorithms have been developed to partition a DAG into a minimal number of chains to facilitate transitive closure computation [11, 5]. Our approach can utilize any of these approaches.

3.2 Transitive Closure between Two Chains

In this work, we will derive a more concise representation for the transitive closure using the chain decomposition. We base this representation on a key observation on how the transitive closure is recorded in binary matrix format. Note that our approach does not need to materialize this binary matrix representation of the transitive closure.

Let M be the binary matrix representation of transitive closure for G. Then, $M[v_i, v_j] = 1$ iff $v_i \to v_j$. $M[v_i, v_j] = 0$ iff v_i cannot reach v_j. We define an index (i, j) for M to be a *cell*. If $M(i, j) = 0$, we say (i, j) is a *0-cell*; else (i, j) is a *1-cell*. Also, we order the vertices based on their chain ID and within each chain, we sort the vertices according to their order ID (oid). Thus, the vertices in the same chain are contiguous in a linearly increasing order. We also introduce the sub-matrix for any two chains C_i and C_j as M_{C_i,C_j}, which has the rows of C_i and columns of C_j. Clearly, the complete transitive closure M can be written as the union of the $k \times k$ submatrices, such as

$$M = \begin{bmatrix} M_{C_1,C_1} & M_{C_1,C_2} & \cdots & M_{C_1,C_k} \\ M_{C_2,C_1} & M_{C_2,C_2} & \cdots & M_{C_2,C_k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{C_k,C_1} & M_{C_k,C_2} & \cdots & M_{C_k,C_k} \end{bmatrix} \quad (1)$$

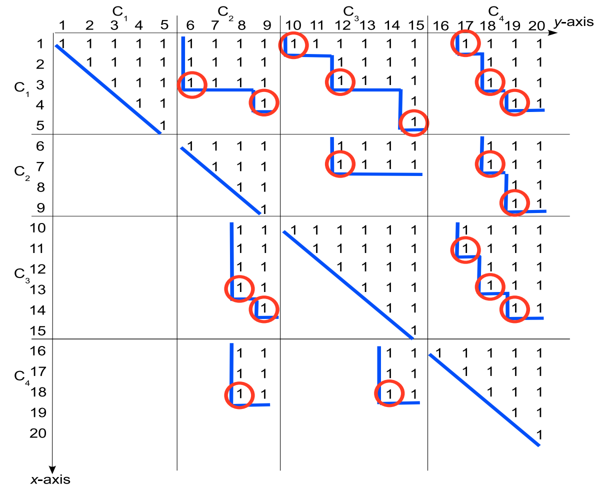

Figure 4: Pseudo-diagonal and Pseudo-upper triangular submatrix. All blank cells are 0-cells.

It is easy to see that any M_{C_i,C_i} is a special upper triangular matrix, i.e., for any $v_a \preceq v_b$ where v_a and v_b are vertices of chain C_i, $M[v_a, v_b] = 1$ and for any $v_a > v_b$, $M[v_a, v_b] = 0$. We refer to it as an *upper uni-triangular matrix*. Note that the geometry of this submatrix describes and is equivalent to the intra-chain reachability property. Therefore, there is no need to materialize M_{C_i,C_i} upper uni-triangular matrices. Next, what does a submatrix M_{C_i,C_j} look like when $i \neq j$?

To describe the shape of the submatrices between any two paths, we introduce the following notation. Given submatrix M_{C_i,C_j} with $|C_i|$ rows and $|C_j|$ columns, and two cells (x, y) and (x', y') where x and x' are vertices of chain C_i and y and y' are vertices of chain C_j, we say *cell (x, y) dominates cell (x', y') in the matrix M_{C_i,C_j} if $x \succeq x'$ and $y \preceq y'$*. In other words, a cell dominates all the cells

located in its upper-right quadrant. As a simple observation, in any submatrix M_{C_i,C_j}, the collection of all the cells being dominated by cell (x, y) form a rectangle which has (x, y) as its lower-left corner and upper right cell of M_{C_i,C_j} as its upper-right corner.

DEFINITION 2. **(Pseudo-Diagonal and Pseudo-Upper Triangular matrix** *The pseudo-diagonal of a binary matrix (submatrix) M_s is a set of* 1*-cells, such as* $\{(x_1, y_1), (x_2, y_2), \cdots, (x_l, y_l)\}$, *such that 1) all the* 1*-cells in M_s are dominated by at least one pseudo-diagonal cell, 2) none of the* 0*-cells in M_s are dominated by any pseudo-diagonal cell, and 3) no pseudo-diagonal cell dominates another pseudo-diagonal cell. If a binary matrix (submatrix) has a pseudo-diagonal, we refer to it as a* pseudo-upper triangular matrix (submatrix).

Clearly, not every binary matrix is a pseudo-upper triangular matrix containing a pseudo-diagonal. We next provide the following theorem to reveal the shape of a submatrix between two chains.

THEOREM 1. *Let M_{C_i,C_j} be the binary submatrix of the transitive closure between two different chains, C_i and C_j. M_{C_i,C_j} is a pseudo-upper triangular matrix.*

Proof Sketch: Our proof is constructive. We will first construct the pseudo-diagonal explicitly. Then, we will show that the matrix is indeed pseudo-upper triangular. Let the chain C_i be $(v_1, v_2, \cdots, v_p)$. Let $f(v_i)$ be the first vertex in C_j v_i can reach. If v_i does not reach any vertex in C_j, let $f(v_i) = +\infty$.

Then we construct the sequence as $(f(v_1), f(v_2), \cdots, f(v_p))$. We can show $f(v_i) \preceq f(v_{i+1})$ as follows: Because v_i reaches v_{i+1}, v_i will reach $f(v_{i+1})$. Thus, $f(v_i)$ should be no larger than $f(v_{i+1})$. This also suggests that $f(v_i) = +\infty$, if exists, can only appear at the end of a sequence.

Given this, we observe the following property for pseudo-diagonal: A 1-cell $(v_i, f(v_i))$ $(1 \leq i \leq p-1)$ is in the pseudo-diagonal if and only if $f(v_{i+1}) \succ f(v_i)$ and $f(v_i) \neq +\infty$. Besides, $(v_p, f(v_p))$ is in the pseudo-diagonal if and only if it is a 1-cell. Thus, we can scan the sequence $(f(v_1), f(v_2), \cdots, f(v_p))$ once to create the pseudo-diagonal.

Now, we only need to show that any cell which is dominated by one of the cells in the pseudo-diagonal is a 1-cell and otherwise, a 0-cell. Let (a, b) be a cell in the matrix and assume it is dominated by one of the cell in the pseudo-diagonal, $(v_i, f(v_i))$. Then, by definition, $a \preceq v_i$, and $b \succeq f(v_i)$. In other words, $a \preceq v_i$ in C_i and $f(v_i) \preceq b$ in C_j. We also know $v_i \rightarrow f(v_i)$. Thus, we have $a \rightarrow b$, so (a, b) is a 1-cell.

Let (c, d) be a cell in the matrix and assume it is not dominated by any of the cells in the pseudo-diagonal. Basically, we have $d \prec f(c)$. Since $f(c)$ is the smallest vertex in chain C_j c can reach, then c cannot reach d, meaning (c, d) is a 0-cell. □

In Figure 4, we can see each M_{C_i,C_j}, $i \neq j$, is a pseudo-upper triangular matrix. We highlight their pseudo-diagonal cells with a circle.

COROLLARY 1. *The transitive closure from any chain C_i to another chain C_j,* $\begin{bmatrix} M_{C_i,C_i} & M_{C_i,C_j} \\ \emptyset & M_{C_j,C_j} \end{bmatrix}$ *can be described as a directed graph, with vertex set $V = V(C_i) \cup V(C_j)$ and edge set $E = E(C_i) \cup E(C_j) \cup \{(v_i, f(v_i)) | (v_i, f(v_i))$ is a pseudo-diagonal cell$\}$, and no two edges cross, i.e., for any two pseudo-diagonal cells, $(v_i, f(v_i))$ and $(v_j, f(v_j))$, we have either*
$(v_i.oid > v_j.oid) \wedge (f(v_i).oid > f(v_j).oid)$, or
$(v_i.oid < v_j.oid) \wedge (f(v_i).oid < f(v_j).oid)$.

Essentially, the edge links from C_i to C_j do not *cross* each other. Figure 5 shows two examples: edge links between C_1 and C_3, and edge links between C_3 and C_4. Moreover, we can see that for chains C_i and C_j, the starting vertices of the pseudo-diagonal cells naturally divide chain C_i into several "outgoing" segments such that all the vertices in a segment share the same "entry point" to chain C_j. Similarly, the end vertices of these pseudo-diagonal cells can divide chain C_j into several "incoming" segments where all the vertices in a segment share the same "exit point" from chain C_i. For instance, in Figure 5, for chain C_3 and C_4, the pseudo-diagonal cells, $\{(11, 17), (13, 18), (14, 19)\}$ divide chain C_3 into four outgoing segments, $(10, 11)$, $(12, 13)$ and $(14, 14)$, and chain C_4 into three incoming segments, $(17, 17)$, $(18, 18)$ and $(19, 20)$.

Now, we formally introduce the *transitive closure contour*.

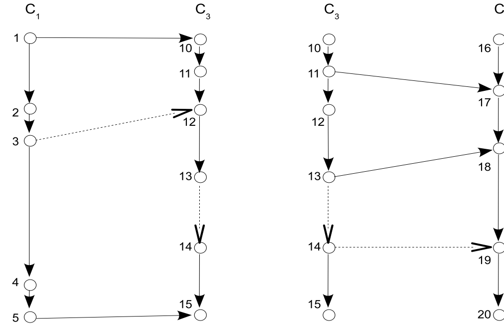

Figure 5: Edgelink between chain C_1 and C_3, and between chain C_3 and C_4. Dotted arrows are virtual edges (paths).

DEFINITION 3. **(Transitive Closure Contour)** *Given DAG G and its chain-decomposition, $C_1 \cup C_2 \cup \cdots C_k$, the transitive closure contour, $Con(G)$ is the set of all pseudo-diagonal cells for each pseudo-upper triangular matrix, M_{C_i,C_j}, where $i \neq j$.*

Given a chain decomposition, we can see the transitive closure contour can precisely describe the complete transitive closure. We will utilize this concise representation of transitive closure to build our 3-HOP indexing.

3.3 Computing Transitive Closure Contour

We now present an efficient computation which can directly compute the transitive closure contour without materializing the binary matrix given a chain decomposition. The sketch of *TransitiveClosureContour* is in Algorithm 1. We use a matrix S to record the entire transitive closure contour of DAG G, $Con(G)$. Each element $S_{i,j}$ records the pseudo-diagonal of M_{C_i,C_j} for chain C_i and C_j.

The computation follows the reverse topological order (Loop 3-21), which broadcasts the reachability information from bottom to top. $S_{i,j}$ is an ordered set of pseudo-diagonal cells (p, q) between chain i and chain j (in ascending order of $q.oid$), and $S_{i,j}.head()$ gets the first (with smallest $q.oid$) pseudo-diagonal cell (p, q) in $S_{i,j}$ For each vertex u, we use $minoid[i]$ to record the smallest vertex it can reach in chain C_i. At the beginning, we fill $minoid[i]$ with the smallest vertex that its own chain $C_{u.cid}$ can reach in chain C_i. This is done in Line 4, and we can retrieve this cell by

Algorithm 1 TransitiveClosureContour($G, C_1 \cup C_2 \cup \cdots \cup C_k$)

Parameter: $C_1 \cup C_2 \cup \cdots \cup C_k$: the Chain Decomposition

1: Perform the Topological Sort of G
2: For each i, j, $1 \le i, j \le k$, $S_{i,j} \leftarrow \emptyset$
3: **for** $u = |V(G)|$ downto 1 {following the reverse topological order} **do**
4: For each i, $1 \le i \le k$, $minoid[i] \leftarrow y$, where $y = q.oid$ and $(p, q) \leftarrow S_{u.cid,i}.head()$ {$y = \infty$ if $S_{u.cid,i} = \emptyset$}
5: **for each** v: the immediate successor of u {in topological order} **do**
6: **if** $v.oid < minoid[v.cid] \wedge v.cid \neq u.cid$ **then**
7: $minoid[v.cid] \leftarrow v.oid$
8: **for each** $i = 1$ to k **do**
9: Let $y = q.oid$: $(p, q) \leftarrow argmin_{(p,q) \in S_{v.cid,i}}(p.oid \ge v.oid)$
10: **if** $u.cid \neq i \wedge minoid[i] > y$ **then**
11: $minoid[i] \leftarrow y$
12: **end if**
13: **end for**
14: **end if**
15: **end for**
16: **for each** $i = 1$ to k **do**
17: **if** $i \neq u.cid \wedge minoid[i] < y$ {$y = q.oid$ and $(p, q) \leftarrow S_{u.cid,i}.head()$} **then**
18: $S_{u.cid,i} \leftarrow S_{u.cid,i} \cup \{(u, minoid[i])\}$
19: **end if**
20: **end for**
21: **end for**

$S_{u.cid,i}.head()$. If $S_{u.cid,i}.head()$ is still empty, we fill $minoid[i]$ with ∞.

After that, we visit each of vertex u's immediate successors, v (Line 5). Our visit follows their topological order, i.e., the smallest vertex will be visited first. Note that by following this order, when we have more than one immediate successor of u in the same chain, we only need to visit the smallest vertex among them (Line 6). Given this, the major operation is to *update the smallest vertices which u can reach using vertex v on each chain*, i.e., to update each $minoid[i]$. Such an update comes from two sources: the first source is from v itself. If $v.oid$ has a smaller sequence number than the current $minoid[v.cid]$, meaning the edge (u, v) allows u to reach a smaller vertex on v's chain; the second source is from the smallest vertices on other chains which v can reach. In the latter case, for each chain C_i (Line 8), we need to get a pseudo-diagonal cell (p, q) in $S_{C_{v.cid},i}$, where p and v are in the same chain and p is the smallest vertex v dominates (Line 9). Thus, q is the smallest vertex u can reach via edge (u, v) directly. Given this, we will test if q is smaller than the current smallest vertex u can reach in chain i, and replace it if it does (Line 10). Finally, after visiting all u's immediate successors, we will add cell $(u, minoid[i])$ to $S_{u.cid,i}$ if it is a pseudo-diagonal cell (Line 17 and 18).

The correctness of Algorithm 1 can be observed by the fact that we maintain a list of the smallest vertex of each chain vertex u can reach in $minoid$, and a cell $(u, minoid[i])$ is a pseudo-diagonal cell iff $minoid[i]$ is less than the smallest vertex in $S_{u.cid,i}$ (Corollary 1). The time complexity of this algorithm is $O(mk \log n)$ in the worst case. This is because the two biggest *for* loops, i.e. step 3 to step 21, run m times, since DAG G has m edges, and the loop from step 8 to 13 runs k time and finally step 9 takes $O(\log n)$ time to do the binary search in the worst case.

4. 3-HOP LABELING FOR TRANSITIVE CLOSURE CONTOUR

4.1 Problem Definition

Our goal in this section is to compress the transitive closure contour, $Con(G)$, using the 3-hop strategy. For any vertex pair $(u, v) \in Con(G)$, we say u is an *out-anchor* vertex for the contour, and v is an *in-anchor* vertex. We will assign each out-anchor vertex a list of intermediate "entry points" of some chains and assign each in-anchor vertex a list of intermediate "exit points" of some chains. To recover the reachability between an out-anchor u and an in-anchor v, we will see if u can reach v in three hops, i.e., the first hop from u to an intermediate entry point, the second hop to the intermediate exit point, and the third hop from the exit point to v. Formally, we introduce the 3-hop reachability labeling for the contour set $Con(G)$ as follows.

DEFINITION 4. **(3-HOP Reachability Labeling)** *Let $Con(G)$ be the transitive closure contour for G with respect to a chain-decomposition. Let V_{out} and V_{in} be the sets of out-anchor vertices and in-anchor vertices for $Con(G)$, respectively. A 3-hop reachability labeling assigns each out-anchor vertex u in V_{out} a label $L_{out}(u)$ (a set of intermediate entry points), and each in-anchor vertex v in V_{in} a label $L_{in}(v)$ (a set of intermediate exit points), such that $L_{out}(u), L_{in}(v) \subseteq V(G)$, and for every $x \in L_{out}(u)$, $u \rightarrow x$ and for every $y \in L_{in}(v)$, $y \rightarrow v$. Furthermore, we have the following two conditions:*

$$(1) \quad (u, v) \in Con(G) \Longrightarrow \exists x \in L_{out}(u), \exists y \in L_{in}(v), \text{ such that } x, y \in C_i, \text{ and } x \preceq y$$

$$(2) \quad \text{for any } x \in L_{out}(u), y \in L_{in}(v), x, y \in C_i, \text{ and } x \preceq y \Longrightarrow u \rightarrow v$$

The size of the labeling is defined to be

$$Cost(3hop) = \sum_{u \in V_{out}} |L_{out}(u)| + \sum_{v \in V_{in}} |L_{in}(v)|$$

To simplify our discussion, we assume $u \in L_{out}(u)$ and $v \in L_{in}(v)$.

THEOREM 2. *Finding a minimum 3-hop reachability labeling for a given contour set $Con(G)$ of a DAG G is an NP-hard problem.*

Proof Sketch: We simply note that 3-hop labeling is a generalization of 2-hop labeling. □

To better understand this problem, we will describe it as a generalized "factorization" problem and then transform it to the classical set-cover problem. We start by partitioning each of the two anchor sets, V_{out} and V_{in}, according to their intermediate chains:

$$V_{out}^i = \{u | u \in V_{out} \text{ and } L_{out}(u) \cap C_i \neq \emptyset\}$$
$$V_{in}^i = \{v | v \in V_{in} \text{ and } L_{in}(v) \cap C_i \neq \emptyset\}$$

Basically, V_{out}^i contains those out-anchor vertices which record intermediate vertices (entry points) in chain C_i. Similarly, V_{in}^i contains those in-anchor vertices which record intermediate vertices (exit points) in chain C_i. Further, for each $u \in V_{out}^i$, we define $L_{out}^i(u)$ to be the vertex of $L_{out}(u) \cap C_i$, and for each $v \in V_{in}^i$ we define $L_{in}^i(v)$ to be the vertex of $L_{in}(v) \cap C_i$. By Corollary 1, $L_{out}(u) \cap C_i$ (or $L_{in}(v) \cap C_i$) contains at most one vertex. Given

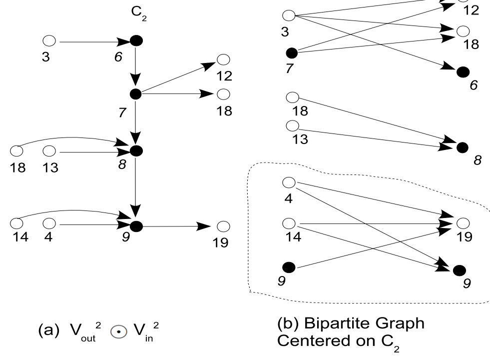

Figure 6: Generalized Join and Chain-Center Bipartite Graph

this, we introduce the following generalized *join* operator (similar to Cartesian product):

$$V_{out}^i \odot V_{in}^i = \{(u,v)|u \in V_{out}^i, v \in V_{in}^i, L_{out}^i(u) \preceq L_{in}^i(v)\}$$

In Figure 6(a), assume all the vertices on the left of chain C_2 record their corresponding entry points into chain C_2, and all the vertices on the right record their exit points. For $v = 12, 18, 19$, assume $L_{in}(v) \cap C_2 \neq \emptyset$. For $u = 3, 18, 13, 14, 4$, assume $L_{out}(v) \cap C_2 \neq \emptyset$. Then, $V_{out}^2 = \{3, 18, 13, 14, 4\}$ and $V_{in}^2 = \{12, 18, 19\}$, and $V_{out}^2 \odot V_{in}^2$ contains all the vertex pairs (u, v), where u is on the left and v is on the right, such that u can reach v via the edges in the graphs, i.e., $\{(3, 12), (3, 18), (3, 19), \cdots, (4, 19)\}$. It also contains all the edges in the graph, i.e., $\{(3, 6), (7, 12), \cdots, (9, 19)\}$.

We consider $\{V_{out}^1, \cdots V_{out}^k\} \otimes \{V_{in}^1, \cdots V_{in}^k\} = V_{out}^1 \odot V_{in}^1 \cup \cdots \cup V_{out}^k \odot V_{in}^k$ to be generalized factorization. Hence, we define the cost of factorization as follows:

$$Cost(factorization) = \sum_{i=1}^{k} |V_{out}^i| + \sum_{i=1}^{k} |V_{in}^i|$$

Given this, we can rewrite our 3-hop reachability labeling problem as a generalized "factorization" problem: *By assigning label $L_{out}(u)$ for each vertex $u \in V_{out}$ and $L_{in}(v)$ for each vertex $v \in V_{in}$, we want to find a factorization $\{V_{out}^1, \cdots V_{out}^k\} \otimes \{V_{in}^1, \cdots V_{in}^k\}$ with minimum cost such that*

$$Con(G) \subseteq V_{out}^1 \odot V_{in}^1 \cup \cdots \cup V_{out}^k \odot V_{in}^k$$

It is easy to see that the 3-hop reachability labeling problem is equivalent to the generalized factorization of $Con(G)$ where the 3-hop indexing cost is equivalent to the corresponding factorization cost:

$$Cost(3hop) = Cost(factorization)$$

In the following subsections, we will derive efficient algorithms to produce minimized factorization and thus also the minimized 3-hop labeling.

4.2 A Basic Approximation Algorithm for 3-Hop Cover

In this subsection, we will transform the factorization problem into a set-cover problem. For this purpose, we will first introduce the notation of the *chain-center bipartite graph*.

DEFINITION 5. **(Chain-Center Bipartite Graph)** *Given a DAG G and a chain decomposition, $C_1 \cup C_2 \cup \cdots \cup C_k$, we construct the chain-center bipartite graph for each chain as follows. Let $\mathcal{B}_i = (\mathcal{X}_i \cup \mathcal{Y}_i, \mathcal{E}_i)$ be the chain-center bipartite graph:*

$\mathcal{X}_i = \{u | \exists a \in C_i,$ *such that* $(u, a) \in Con(G)\} \cup \{b | b \in C_i,$ *such that* $\exists v, (b, v) \in Con(G)\}$

$\mathcal{Y}_i = \{v | \exists b \in C_i,$ *such that* $(b, v) \in Con(G)\} \cup \{a | a \in C_i,$ *such that* $\exists u, (u, a) \in Con(G)\}$

$\mathcal{E}_i = \{(x, y) | x \in \mathcal{X}_i$ *and* $y \in \mathcal{Y}_i$ *and* $(x, y) \in Con(G)\}$

Figure 6(b) is an example showing the bipartite graph for chain C_2. Now we can transform the factorization problem into the set-cover problem as follows: Let the grounding set be $Con(G)$. Let the set of candidates be $\{\hat{B}_i | \hat{B}_i$ is a subgraph of $\mathcal{B}_i$ where $1 \leq i \leq k\}$. The weight of a candidate bipartite subgraph should reflect the related index cost which is defined as the number of vertices in $V(\hat{B}_i)$, i.e., $weight(\hat{B}_i) = |V(\hat{B}_i)|$. For example, in Figure 6, the circled bipartite subgraph has weight 5.

Then we may apply the classical greedy algorithm [8] to find the minimal set cover as follows. *Let R be the uncovered contour pairs (initially, $R = Con(G)$). For each candidate set $\hat{B}_i$, where the vertex sets $X(\hat{B}_i) \subseteq \mathcal{X}_i$ and $Y(\hat{B}_i) \subseteq \mathcal{Y}_i$ and edge set $E(\hat{B}_i) \subseteq \mathcal{E}_i$, we define the compression ratio of selecting $\hat{B}_i$ as*

$$\rho(\hat{B}_i) = \frac{|E(\hat{B}_i) \cap R|}{weight(\hat{B}_i)} = \frac{|E(\hat{B}_i) \cap R|}{|X(\hat{B}_i)| + |Y(\hat{B}_i)|}$$

At each iteration, the greedy algorithm selects the candidate set with the highest compression ratio and puts it in the resulting set. Then, the algorithm will update R by removing the newly covered contour pairs, $R = R \backslash E(\hat{B}_i)$. The procedure proceeds until all contour pairs are covered (i.e. $R = \emptyset$).

It has been proved that the approximation ratio of this algorithm is $ln(|Con(G)|) + 1$ [8]. We now link this problem and its results back to the aforementioned factorization problem. First, we note that picking up a subgraph $\hat{B}_i$ in the set cover corresponds to adding a generalized join between $X(\hat{B}_i)$ and $Y(\hat{B}_i)$, i.e., $X(\hat{B}_i) \odot Y(\hat{B}_i)$. This is because each non-C_i vertex v in $\hat{B}_i$ needs to record in $L_{out}(v)$ an entry point to chain C_i, or record in $L_{in}(v)$ an exit point from chain C_i. It is easy to observe that non-C_i vertices account for at least half in $\hat{B}_i$. Given such a labeling, we can guarantee to cover all the edges of $E(\hat{B}_i)$, i.e., $X(\hat{B}_i) \odot Y(\hat{B}_i) \supseteq E(\hat{B}_i)$. Here, we may produce some edges which do not belong to the contour, but this will not affect set cover results. Indeed, in the factorization formulation, we may also produce extra edges which do not belong to the contour. However, those edges all belong to the complete transitive closure and thus will not affect the correctness of our reachability indexing.

Second, we note that the optimal set cover results will choose at most one subgraph from each chain-center bipartite graph, i.e., each vertex in each bipartite graph will be selected only once. In the greedy algorithm, we may find several subgraphs which all come from the same bipartite graphs. In this case, we can simply combine their label sets, and the weight of the resulting subgraphs will be no higher than the sum of the weights of these individual subgraphs. Thus, this optimal result of the set-cover problem can be rewritten exactly as a factorization result with each chain having at most one join centered on it, and our approximation bound is maintained.

However, the major issue here is that the number of candidate subgraphs is exponential. A similar issue exists for 2-hop labeling. As suggested in [9], we can deal with this problem by realizing that finding $\hat{B}_i$ of the highest compression ratio is equivalent to finding the densest subgraph of the bipartite graph $\mathcal{B}'_i = (\mathcal{X}_i \cup \mathcal{Y}_i, \mathcal{E}_i \setminus R)$.

Given this, the basic idea of 3-hop labeling algorithm is: *for each iteration, we first find the densest subgraph of each bipartite graph $\mathcal{B}'_i$, and then among them (k subgraphs), we choose the densest one and update the set R of uncovered contour pairs. We repeat this iteration until R is empty.*

Since finding the densest subgraph forms the core of our 3-hop labeling algorithm, we formulate it precisely here:

DEFINITION 6. **(Densest Subgraph Problem)** *Let $G = (V, E)$ be a graph (directed or undirected). For any subset $V_s \subseteq V$, let $G[V_s] = (V_s, E_s)$ be the induced subgraph of G, i.e., $E_s = E \cap V_s \times V_s$. The densest subgraph problem is to find a subset $V_s \subseteq V$, such that the density of the induced subgraph, $d = \frac{|E_s|}{|V_s|}$, $G_s = (V_s, E_s)$, is maximized.*

The fastest exact algorithm for the densest subgraph problem runs in $O(|V||E| \log |V|^2/|E|)$ [10]. In 2-hop labeling [9], the author suggests using a linear 2-approximation algorithm for the densest subgraph problem. Their algorithm is a simple variant of [14]. It iteratively removes a vertex with the minimal degree from the graph, and this gives V subgraphs. It returns a 2-approximate densest subgraph and can run in linear time in the number of edges in the graph.

In the next subsection, we will introduce a new approach to identify the densest subgraph, which will allow us to prune the search space of these candidate subgraphs significantly.

4.3 A Faster Algorithm for 3-HOP Labeling

To describe our new algorithm for densest subgraph discovery, we introduce the *rank subgraph*.

DEFINITION 7. **(Rank Subgraph)** *Let $G = (V, E)$ be an undirected graph. Given a positive integer d, we will remove all the vertices which have degree less than d and their adjacent edges in G, and then we repeat this procedure to the new graph. Let G_d be the resulting subgraph of G where each vertex in G_d is adjacent to at least d other vertices in G_d. If no vertices are left in the graph, we refer to it as the empty graph, denoted as $G_\emptyset$. Given this, we construct a subgraph sequence $G \supseteq G_1 \supseteq G_2 \cdots \supseteq G_l \supset G_{l+1} = G_\emptyset$, where $G_l \neq G_\emptyset$ and contains at least $l+1$ vertices. We define l as the rank of the graph G, and G_l as the rank subgraph of G.*

Given this, we will use G_l as the approximate densest subgraph.

LEMMA 1. *Given G, let G_s be the densest subgraph of G, with density $d(G_s)$, and let G_l be its rank subgraph with density $d(G_l)$. Then, the density of G_l is no less than half of the density of G_s:*

$$d(G_l) \geq \frac{d(G_s)}{2}$$

Proof Sketch: We prove this by way of contradiction. Suppose $d(G_l) < \frac{d(G_s)}{2}$, which suggests $d(G_s) > 2 \times d(G_l) = 2\frac{|E(G_l)|}{|V(G_l)|} \geq 2\frac{l|V(G_l)|/2}{|V(G_l)|} = l$

Then, we claim that each vertex in G_s should have degree more than l, i.e., for any $v \in V(G_s)$, $degree(v) > l$. If not, assume $v' \in V(G_s)$ has vertex degree $d_{v'} \leq l$. Then, we can simply remove this vertex to increase the density of the subgraph: $\frac{|E(G_s)|-d_{v'}}{|V(G_s)|-1} = \frac{d(G_s)|V(G_s)|-d_{v'}}{|V(G_s)|-1} > \frac{d(G_s)|V(G_s)|-d(G_s)}{|V(G_s)|-1} = d(G_s)$

Since each vertex in G_s has degree more than l, we conclude that $G_s \subseteq G_{l+1}$. However, $G_{l+1} = G_\emptyset$, which contradicts that there is a G_s with density more than $2 \times d(G_l)$. □

Following this, we have the following interesting observation.

THEOREM 3. *Consider we have k bipartite graphs, $\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_k$. Let $l_1, l_2, \cdots, l_k$ be their respective ranks. Let $S_1, S_2, \cdots, S_k$ be their respective densest subgraphs, let $G_{l_1}(\mathcal{B}_1)$, $G_{l_2}(\mathcal{B}_2)$, $\cdots, G_{l_k}(\mathcal{B}_k)$ be their respective rank graphs, and let $l_{max} = \max(l_1, l_2, \cdots, l_k)$. Assume we have several maximum rank graphs with rank l_{max}. Then, we claim that any maximum rank graph $G_{l_i}(\mathcal{B}_i)$ where $l_i = l_{max}$ has a density no less than half of the density of the maximal density subgraphs:*

$$d(G_{l_i}(\mathcal{B}_i)) \geq \frac{\max_{1 \leq j \leq k} d(S_j)}{2}$$

Proof Sketch: The proof is similar to Lemma 1. We prove this by way of contradiction. Suppose $d(G_{l_i}(\mathcal{B}_i)) < \frac{\max_{1 \leq j \leq k} d(S_j)}{2}$. Then we can derive

$$\max_{1 \leq j \leq k} d(S_j) > 2d(G_{l_i}(\mathcal{B}_i)) = 2\frac{|E(G_{l_i}(\mathcal{B}_i))|}{|V(G_{l_i}(\mathcal{B}_i))|}$$
$$\geq 2\frac{l_i|V(G_{l_i}(\mathcal{B}_i))|/2}{|V(G_{l_i}(\mathcal{B}_i))|} = l_i = l_{max}$$

Suppose $d(S_p) = \max_{1 \leq j \leq k} d(S_j)$. Then with similar argument as in the proof of lemma 1 we claim all vertices in S_p have degree more than l_{max}. Hence we conclude $S_p \subseteq G_{l_{max+1}}(B_p)$. However, $G_{l_{max+1}}(B_p) = \emptyset$ according to the definition of rank graph, a contradiction. □

The key implication from Theorem 3 is that we can organize all the bipartite graphs in a queue based on their ranks. If we know l is the highest rank of all the bipartite graphs, then we can return the first rank subgraphs we find from these bipartite graphs as the 2-approximation densest subgraph. We employ this technique in the greedy algorithm by deriving an efficient *incremental* search procedure for the densest subgraph from those bipartite graphs at every iteration.

Algorithm 2 3HOPContour(G,$Con(G)$, $C_1 \cup \cdots C_k$)

```
1: Construct Bipartite Graphs B_1, ..., B_k;
2: For each B_i, construct vertex rank groups, compute the rank r_i
   of B_i and the density d_i of the rank graph G_{r_i}(B_i);
3: Sort all B_i into queue Q according to descending order of r_i.
4: R ← Con(G);
5: Pop the first element B from the queue Q;
6: while R ≠ ∅ do
7:    while B.r < B'.r {B' (B' ← Q.pop()) is the next element
      in Q after popping the last bipartite graph} do
8:       B'.r ← RankSubgraph(B', R, B.r)
9:       if B.r < B'.r then
10:         insert B back to Q in the sorting order;
11:         B ← B'
12:      else
13:         insert B' back to Q in the sorting order;
14:      end if
15:   end while
16:   R ← R\E(G_r(B));
17:   Update L_out and L_in for vertices in selected G_r(B).
18:   B.r ← RankSubgraph(B,R,0);
19: end while
```

The sketch of our 3-hop labeling construction algorithm *3HOP-Contour* is in Algorithm 2. It starts with constructing k bipartite graphs, each corresponding to a chain in 3-hop. Initially, we directly compute the rank of each bipartite graph and the density of its corresponding rank subgraph (Line 2). We will then sort all bipartite graphs based on their rank and put them in queue Q

(Line 3). Our goal is to cover the entire transitive closure contour $R = Con(G)$. The algorithm will iteratively pick the densest subgraphs and remove their edges until all the edges (vertex pairs) in the transitive closure contour are covered ($R = \emptyset$). During this covering process, we can make the following observation for the rank of each bipartite graph: *for any bipartite graph, its rank will not be able to increase during the covering process*. This is because in our covering processing, an increasing number of edges in the contour will be covered and similarly for the edge set of each bipartite graph. Say at a certain iteration, we compute the rank for a bipartite graph $\mathcal{B}$, denoted as $\mathcal{B}.r$. Then, if we try to reevaluate its rank for the updated graph, where the edge set is $E(\mathcal{B}) \cap R$, we know the updated rank cannot exceed its earlier rank $\mathcal{B}.r$. Indeed, we can apply $\mathcal{B}.r$ as an upper bound of $\mathcal{B}$'s new rank.

To further speed up the rank subgraph searching procedure, we organize the vertices of each bipartite graph into different *rank groups*: For a given bipartite graph with a rank l, let G_d be the resulting subgraph of a given bipartite graph as we iteratively remove all the vertices with degree less than d. Thus, we have a subgraph sequence, $G \supseteq G_1 \supseteq G_2 \cdots \supseteq G_l \supset G_{l+1} = G_{\emptyset}$. We assign each vertex v a *rank* d, if $v \in V(G_d)$ and $v \notin V(G_{d+1})$. Given this, all the vertices with the same rank will be organized together in each bipartite graph. We note that *the rank of each vertex will not be able to increase during the covering process* as well. Thus, using this organization, we can quickly prune the vertices with rank lower than a given threshold. This will be applied to facilitate the rank graph searching procedure.

The major iteration of our algorithm is in the loop in lines 6 to 19. In every iteration, we greedily select the densest subgraphs from our k bipartite graphs. This is done using the queue in the while loop from Lines 7 to 15. We visit each bipartite graph according to its order in the queue (Line 7). Let $\mathcal{B}$ be the bipartite graph which has the highest rank among all the visited bipartite graphs for the current iteration. Then, we always extract the first bipartite graph $\mathcal{B}'$ in the queue Q and compare its *saved rank* $\mathcal{B}'.r$, which is the upper bound of its real rank, with $\mathcal{B}$'s real rank $\mathcal{B}.r$.

If $\mathcal{B}'.r \leq \mathcal{B}.r$, we know that the current rank is the highest one all the bipartite graphs can have since all the remaining bipartite graphs in the queue will not have a higher rank than $\mathcal{B}'.r$. Thus, we do the following: 1) we first extract the highest ranked subgraph $G_r(\mathcal{B})$ and apply it to cover R (Line 16); 2) we update L_{out} and L_{in} for vertices in $G_r(\mathcal{B})$; 3) we recompute the rank of $\mathcal{B}$ immediately and use it as the first candidate rank for the next iteration (Line 18).

However, if this is not the case ($\mathcal{B}'.r > \mathcal{B}.r$), we need to check if the true rank of $\mathcal{B}'.r$ is higher than $\mathcal{B}.r$. Here we will apply the vertex rank group organization to speed up the search procedure: since we already have bipartite graph $\mathcal{B}$ with rank $\mathcal{B}.r$, we are not interested in $\mathcal{B}'$ if it has equivalent or lower rank. Thus, we invoke the *RankSubgraph* procedure with three parameters: $\mathcal{B}'$ is the targeted bipartite graph, R is the uncovered edges, and the last parameter is the minimal rank in which we are interested. In this case, we are only interested in ranks higher than $\mathcal{B}.r$. This procedure will apply R to remove those edges not in R and update the vertex rank group. Again, it only updates those vertices with rank higher than $\mathcal{B}.r$. This is done in Line 8. For brevity, we omit the details of the *RankSubgraph* procedure.

Putting all of these together, we can see Algorithm 2 creates L_{out} and L_{in} for the out-anchor and in-anchor vertices of the transitive closure contour $Con(G)$. As an example, in Figure 6 one of the densest bipartite subgraphs is the circled subgraph which could be selected by Algorithm 2. If selected, Algorithm 2 will add 9 to $L_{out}(4)$, $L_{out}(14)$, and $L_{in}(19)$. A complete labeling sets L_{in} and L_{out} from Algorithm 2 are shown in Figure 7, where we only show $L_{in}(u)$ (or $L_{out}(u)$) of a vertex u if it is not empty, and set i for L_{in} and set o for L_{out}.

Finally, we can claim the following optimalities of our *3HOP-Contour* algorithm. Due to space constraints, we omit the proofs.

THEOREM 4. *The* 3HOPContour *algorithm finds a* 3*-hop labeling for the transitive closure contour $Con(G)$ whose size is larger than the smallest such labeling by at most an $O(\ln |Con(G)| + 1) = O(\log n)$ factor, where n is the number of vertices in G.*

THEOREM 5. *For any DAG G, the minimum 3-hop labeling cost (defined previously as Cost(3hop)) for transitive closure contour $Con(G)$, Opt_{3-hop}, is always no larger than the minimum labeling cost of 2-hop, Opt_{2-hop}. In addition, the upper bound of 3-hop labeling cost produced by* 3HOPContour *algorithm, $O((\ln |Con(G)| + 1)(Opt_{3-hop}))$, is always no larger than $O((\ln |V|^2 + 1)(Opt_{2-hop}))$, the upper bound of labeling cost produced by Cohen et al's 2-hop algorithm [9].*

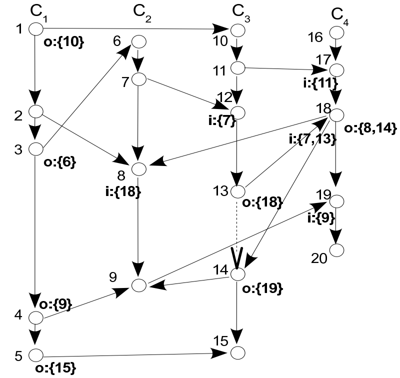

Figure 7: 3**-Hop Labeling of Transitive Closure Contour**

5. REACHABILITY QUERY PROCESSING USING 3-HOP INDEXING

In Section 4, we show how to construct the 3-hop labeling for the transitive closure contour. As a result of Algorithm 2, we get $L_{out}(u)$ and $L_{in}(v)$ for each out-anchor vertex u and each in-anchor vertex v, respectively. In this section we will show how to efficiently answer reachability queries using these labelings. We describe two approaches: the first approach directly applies the 3-hop labeling of the contour to achieve a worst-case time complexity $O(|Con(G)|)$ while the second approach utilizes segments to reduce the query processing complexity.

5.1 3-HOP Contour Query Processing

Note that the 3-hop labeling of the transitive closure contour $Con(G)$ ensures that the reachability for any pair of vertices in a DAG G can be inferred. This is because 3-hop labeling can cover all the vertex pairs in $Con(G)$, and $Con(G)$ can cover all the other vertex pairs in the transitive closure matrix.

Given this, to tell if vertex u in chain C_i can reach vertex v in chain C_j, we can first recover the pseudo-diagonal of M_{C_i,C_j} using

the 3-hop labeling and then test if (u, v) is dominated by any of the pseudo-diagonal cells. However, we do not need to consider those pseudo-diagonal cells or the closure vertex pairs whose out-anchor vertex is smaller than u or whose in-anchor vertex is bigger than v. We can integrate these steps together and have the following query processing procedure:

Step 1: In chain C_i, $(u \in C_i)$, we collect all the smallest vertices on any other chain that u can reach through the out-anchor vertex u', $u \preceq u'$: $(L_{out}^{x.cid}(u') = L_{out}(u') \cap C_{x.cid})$

$$X = \{x | x \in \bigcup_{u' \succeq u} L_{out}(u') \text{ AND } x \preceq L_{out}^{x.cid}(u') \text{ for any } u' \succeq u\}$$

Step 2: In chain C_j, $(v \in C_j)$, we collect all the largest vertices on any other chain which can reach v through the in-anchor vertex v', $v' \preceq v$: $(L_{in}^{y.cid}(v') = L_{in}(v') \cap C_{y.cid})$

$$Y = \{y | y \in \bigcup_{v' \preceq v} L_{in}(v') \text{ AND } L_{in}^{y.cid}(v') \preceq y \text{ for any } v' \preceq v\}$$

Step 3: We see if there is an x, y pair, $x \in X$ and $y \in Y$, such that $x.cid = y.cid$ and $x \preceq y$.

Using the highway analogy, we can see the first step collects those entry points u can reach on the intermediate chains, the second step collects those exit points which reach v on the intermediate chains, and the third step checks to see if an entry point can reach an exit point, i.e., if they are on the same chain and the entry point has smaller sequence number than the exit point. Note that the worst-case query processing cost is $O(|Con(G)|)$. This can be observed by the fact that for any out-anchor vertex u, and $v \in L_{out}$, we have $(u, v) \in Con(G)$ (the same for any in-anchor vertex). Thus, the first two steps cost maximally $O(|Con(G)|)$ time and the third step costs $O(k)$, where k is the number of chains in the chain-decomposition.

For example, in Figure 7, to tell whether $u = 2$ can reach $v = 20$, we get set $X = \{6, 15\}$ by checking $L_{out}(3)$,$L_{out}(4)$ and $L_{out}(5)$; and set $Y = \{9, 13\}$ by checking $L_{in}(19)$,$L_{in}(18)$, and $L_{in}(17)$. Since $6(\in X)$ reaches $9(\in Y)$ in C_2, we say u can reach v.

5.2 3-HOP Segment Query Processing

In this subsection, we introduce an indexing method on top of the 3-hop contour labeling to reduce the query processing complexity. The worst-case query processing time is $O(\log n + k)$, where n is the number of vertices in G. We can see the major bottleneck in the first approach is its first two steps. To speed up these steps, the new approach will break each chain into segments. Specifically, for each chain C_i, we will break it into outgoing segments and incoming segments.

We construct the segments for chain C_i with respect to another chain C_j based on the 3-hop contour labeling. Let $Q_{out}(i, j)$ be all the out-anchor vertices of chain C_i which record an intermediate entry point in chain C_j:

$$Q_{out}(i, j) = \{x | x \in C_i, L_{out}(x) \cap C_j \neq \emptyset\}$$

Let $Q_{in}(i, j)$ be all the in-anchor vertices of chain C_j which record an intermediate exit point in chain C_i:

$$Q_{in}(i, j) = \{y | y \in C_j, L_{in}(y) \cap C_i \neq \emptyset\}$$

Then, we can order all the vertices $x_1, \cdots, x_l$ in $Q_{out}(i, j)$ such that $x_1 \preceq x_2 \preceq \cdots \preceq x_l$, $l = |Q_{out}(i, j)|$, and order all the vertices $y_1, \cdots, y_{l'}$ in $Q_{in}(i, j)$ such that $y_1 \preceq y_2 \preceq \cdots \preceq y_{l'}$, $l' = |Q_{in}(i, j)|$. Given this, we construct the outgoing segments for C_i, denoted by their sequence number,

$$(1, x_1.oid), (x_1.oid + 1, x_2.oid), \cdots (x_{l-1}.oid + 1, x_l.oid)$$

and the incoming segments for C_j,

$$(y_1.oid, y_2.oid - 1), (y_2.oid, y_3.oid - 1), \cdots (y_l.oid, C_j.last().oid)$$

For example, in Figure 7, the outgoing segments constructed from $Q_{out}(1, 2)$ are $(1, 3)$ and $(4, 4)$ where $L_{out}^2(1, 3) = 6$ and $L_{out}^2(4, 4) = 9$. The incoming segments constructed from $Q_{in}(3, 4)$ are $(17, 17)$ and $(18, 20)$ where $L_{in}^3(17, 17) = 11$ and $L_{in}^3(18, 20) = 13$.

We say a vertex v is in a segment $S = (x, y)$ (denoted as $v \in S$) if $x \leq v.oid \leq y$. We note that all the vertices in each outgoing segment share the same entry point of chain C_j and all the vertices in each incoming segment share the same exit point of chain C_i. Thus, we assign each outgoing segment (or incoming segment) with a unique vertex on chain C_j (or C_i) as its label.

In the 3-hop segment indexing, we construct these outgoing and incoming segments of each chain with respect to every other chain. Then for all the segments which share the same starting vertex and ending vertex, we combine all their individual labels into $L_{out}(S)$, where S is the combined segment. In addition, to facilitate query processing, we construct an interval tree [3] for all the outgoing segments in a single chain C_i and an interval tree for all the incoming segments in a single chain C_j. Given this, we can see that the new query processing procedure for answering whether u can reach v is as follows:

Step 1: In chain C_i, $(u \in C_i)$, we collect all the outgoing segments which contain u and combine all their labels in X.

$$X = \{x | x \in \bigcup_{u \in S} L_{out}(S) \text{ AND } x \preceq L_{out}^{x.cid}(S) \text{ for any } S \ni u\}$$

Step 2: In chain C_j, $(v \in C_j)$, we collect all the incoming segments which contain v and combine all their labels in Y.

$$Y = \{y | y \in \bigcup_{v \in S} L_{in}(S) \text{ AND } L_{in}^{y.cid}(S) \preceq y \text{ for any } S \ni v\}$$

Step 3: We see if there is an x, y pair, $x \in X$ and $y \in Y$, such that $x.cid = y.cid$ and $x \preceq y$.

The worst case query processing time is $O(\log n + k)$. Though the number of segments could be maximally n^2, the number of segments covering u or v is actually no more than k. The interval tree can return the segments which cover u in $O(\log n + k)$ time. Finally, we note that the extra segments can contribute to an $O(nk)$ storage cost on top of the 3-hop contour labeling.

6. EXPERIMENTAL EVALUATION

In this section, we empirically compare the new 3-hop labeling approach with the state-of-art simple graph covering approach, the path-tree cover and the 2-hop labeling approach, on both synthetic and real data. We also list query time of two classical approaches, breadth-first search and depth-first search as benchmarks. We are particularly interested in the following issues:

1. **Index size:** The major goal of this work is to derive an indexing scheme for reachability query which can significantly compress the transitive closure when the ratio between the number of edges and the number of vertices is relatively high. Specifically, we would like to learn how much we can gain by using 3-hop labeling compared with two best available indexing approaches, path-tree and 2-hop. Since each vertex in the path-tree is labeled by three numbers (two numbers are

tree intervals and one number is depth-first order), and each vertex in the 3-hop is labeled by two numbers (cid and oid), we define the index size of the path-tree scheme for a graph $G = (V, E)$ to be the size of transitive closure plus $3 * |V|$, and the index size of the 3HOP-Contour to be $cost(3hop)$ (defined in subsection 4.1) plus $2 * |V|$. The index size of the 3HOP-Segment is the size of all segments, i.e. two times the number of segments, plus the cost of labeling. In this case each segment has a label instead of each vertex. It is easy to observe that the total labeling cost of 3HOP-Segment is $cost(3hop)$, the same as that of 3HOP-Contour.

2. **Query processing time:** As we mentioned before, there is a trade-off between the compression rate of the transitive closure and the query answering time. In order to achieve a high compression rate, the 3-hop indexing approach clearly requires more runtime processing for answering reachability queries than path-tree. However, the interesting question is how fast 3-hop can answer queries and whether it is comparable with path-tree and 2-hop.

3. **Construction time:** A major advantage of 3-hop compared with 2-hop is that it does not require computing the full transitive closure and employs a new strategy to speedup the densest subgraph identification. How can these factors speedup the labeling process of the 3-hop compared with the 2-hop approach?

Given this, we have specifically compared these six algorithms in the experimental evaluation: 1) the original 2-hop approach by Cohen *et al.* [9], denoted as **2HOP**; 2) the path-tree approach (PTree-1) proposed by Jin *et al.* [12], denoted as **Path-Tree**; 3) the 3-hop labeling approach with 3-hop contour query processing, denoted as **3HOP-Contour**; 4) the 3-hop labeling approach with 3-hop segment query processing, denoted as **3HOP-Segment**; 5) Breadth-first Search; and 6) Depth-first Search. We have implemented all six algorithms. The Path-Tree is an improved second version with respect to the first version in [12]. Besides, since 3-hop needs a chain decomposition, we implement a heuristic algorithm developed by Jagadish, *procedure-3* in [11]. All algorithms are implemented using C++ based on the Standard Template Library (STL). We perform experiments on a Linux 2.6 machine with 2.0GHz CPU and 8.0GB RAM.

In the experiments, we collect all three measures: the index size, the query time, and the indexing construction time, and each experiment processes $100,000$ randomly generated queries.

6.1 Synthetic Datasets

Here, we run two sets of experiments using the synthetic DAGs, which are generated using a random directed acyclic graph generation algorithm described in [13].

In the first experiment, we generate a set of DAGs with $2,000$ vertices, and vary their average density from 2 to 12. We compare all six approaches, 3HOP-Segment, 3HOP-Contour, 2HOP, Path-Tree, breadth-first search, and depth-first search in this experiment.

From figure 8, both 3HOP-Segment and 3HOP-Contour consistently obtain a better index size compression rate than 2HOP and Path-Tree on all synthetic datasets. Overall, the index size of 3HOP-Contour and 3HOP-Segment are on average about 2.7 times and 2.0 times better than the Path-Tree approach, and about 1.5 times and 1.1 times better than 2HOP. On the other hand, in Table 2, we observe that path-tree has moderately faster query time

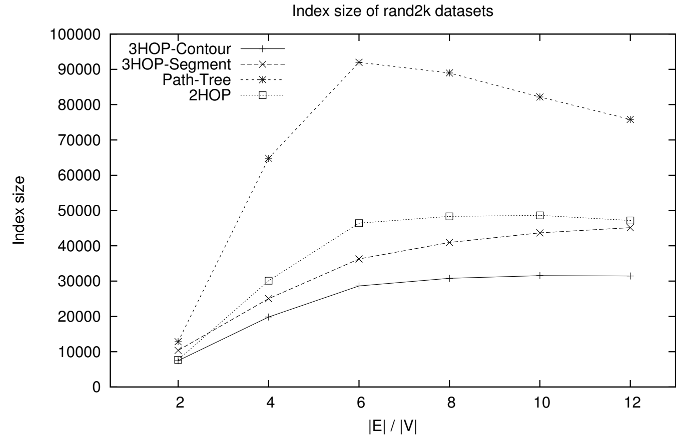

Figure 8: Index size of Synthetic Datasets (2K)

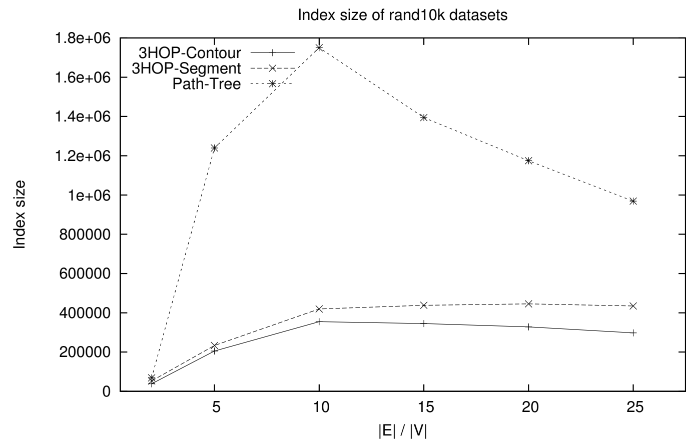

Figure 9: Index size of Synthetic Datasets (10K)

Dataset	Query Time (in ms)					
	3HOP-Contour	3HOP-Segment	Path-Tree	2HOP	Breath-First Search	Depth-First Search
rand2k_2	22.865	165.646	9.108	70.239	891.957	891.502
rand2k_4	49.354	566.175	26.051	297.801	2197.01	1796.84
rand2k_6	77.686	1092.2	33.785	514.546	4397.49	4358.84
rand2k_8	103.769	1422.82	31.626	589.059	6134.99	7553.84
rand2k_10	124.291	1661.82	28.322	574.64	7499.31	11305.3
rand2k_12	141.825	1748.04	28.411	722.005	8628.21	14917.4

Table 2: Query Time of Synthetic Datasets (2K)

Dataset	DAG #V	DAG #E	Density
Arxiv	6000	66707	11.12
Citeseer	10720	44258	4.13
Go	6793	13361	1.97
Pubmed	9000	40028	4.45
Yago	6642	42392	6.38

Table 3: Real datasets

than 3HOP-Contour, as expected. However, 3HOP-Contour has not only smaller index size than that of 3HOP, but shorter query time as shown in Table 2. It is interesting to observe that 3HOP-Contour is faster than 3HOP-Segment. We know that the query time complexity of 3HOP-Segment is better than 3HOP-Contour. However, in practice, we can see intuitively that more memory access operations (e.g. searching interval trees and processing search results) are needed in 3HOP-Segment, and interval trees are too big to be loaded into system caches. Thus, it is reasonable that the query time of 3HOP-Contour is better than that of 3HOP-Segment.

In terms of construction time, 3HOP-Contour and 3HOP-Segment are several orders of magnitude faster than 2HOP. In this experiment, 2HOP finishes index construction of a dataset between 7 and 21 hours, while 3HOP-Segment and 3HOP-Contour take only 1 second to 71 seconds. To explain the phenomena, we notice that 3-hop needs to take $O((kn^2) * |Con(G)|)$ construction time (Recall we have k bipartite graphs corresponding to k chains. Each bipartite graph starts as a complete bipartite graph with $O(n^2)$ edges.) while 2-hop takes $O(n^3|T_c|)$. $|Con(G)|$ is the number of contour points and $|T_c|$ is the size of transitive closure. Although in worst case $|Con(G)|$ could be equal to $|T_c|$, in practice, $|Con(G)|$ is much smaller. In addition, we have developed and implemented a new technique (Theorem 3) which can speed up 3-hop labeling by up to $O(k)$.

In the second experiment, we generate random DAGs with $10,000$ vertices, and vary their densities from 2 to 25. Note that we do not compare with 2HOP in the second experiment because 2HOP cannot process such large scale datasets due to memory constraints. Figure 9 shows the index size of the two 3-hop approaches and the path-tree approach. Here, 3HOP-Contour and 3HOP-Segment can achieve up to 6.0 times and 5.3 times smaller index sizes than Path-Tree. On average, 3HOP-Contour and 3HOP-Segment have 3.9 times and 3.1 times smaller index sizes, respectively, than Path-Tree. The query processing time and construction time are similar to the first experiments and we omit them here.

It is interesting to observe that there is a peak occurring at density 10 on the index size for all three algorithms. Since 3-hop labeling relies on chain decomposition and path-tree labeling depends on path decomposition, an increase in density potentially may result in better chain or path decomposition (i.e. with fewer chains or paths w.r.t. DAG). This can explain the peak phenomena.

6.2 Real Datasets

To evaluate our indexing scheme on real datasets, we have collected five real datasets listed in Table 4. All graphs are extracted from real-world large datasets with density being larger than or close to 2. Among them, arXiv is extracted from a dataset of citations among scientific papers from the arxiv.org website [1]. Similarly, citeseer contains citations among scientific literature publications from the CiteSeer project [2], and pubmed was extracted from an XML registry of open access medical publications from the PubMed Central website [3]. GO contains genetic terms and their relationships from the Gene Ontology project [4]. Yago describes the structure of relationships among terms in the semantic knowledge database from the YAGO project [5].

Table 4 shows the index size and query time of three methods, the two 3-hop approaches and the path-tree approach. Again, in this experiment, the 2HOP approach fails by running out of memory.

As shown in the table, the index sizes of 3HOP-Contour are reduced significantly with respect to Path-Tree, and the index sizes of 3HOP-Segment are smaller than Path-Tree in 3 out of 5 datasets. On average, 3HOP-Contour and 3HOP-Segment obtain 1.7 times and 1.2 times better compression rates than the Path-Tree approach. As expected, we found that the query time of Path-Tree is better than 3HOP.

The 3HOP-Contour has a similar construction time to 3HOP-Segment therefore we only report 3HOP-contour construction time here. It takes 8530, 106, 25, 257, and 25 seconds to construct indexing for dataset arXiv, citeseer, go, pubmed, and yago, respectively. The Path-Tree is much faster and takes only 10, 0.73, 0.2, 0.77, and 0.55 seconds, respectively for these datasets. This is expected since the 3-hop approach is more computationally expensive. However, the new approach has an evidently higher compression rate and its query processing time is also comparable to the path-tree approach.

7. CONCLUSION

In this work, we introduce a new 3-hop indexing scheme with high compression rate targeting the directed graphs with higher edge-vertex ratio. We not only show that our index size can achieve a guaranteed approximation bound, but also demonstrate its applicability through extensive experimental evaluations on both real and synthetic datasets. More importantly, we believe this method potentially opens a new way to compress the transitive closure and leads to new provocative questions. For instance, how can other simple graph structures, such as trees, serve as the intermediate hop (highway)? How can we derive the average complexity of these

[1]http://arxiv.org/

[2]http://citeseer.ist.psu.edu/oai.html

[3]http://www.pubmedcentral.nih.gov/

[4]http://www.geneontology.org/

[5]http://www.mpi-inf.mpg.de/ suchanek/downloads/yago/

Dataset	Index Size			Query Time (in ms)				
	3HOP-Contour	3HOP-Segment	Path-Tree	3HOP-Contour	3HOP-Segment	Path-Tree	Breadth-First Search	Depth-First Search
ArXiv	47472	64378	86855	125.382	1060.2	24.278	19029.2	129587
Citeseer	51035	72167	91820	87.763	523.488	23.32	4567.16	4781.19
Go	27764	41798	37729	53.354	250.261	10.39	2697.67	2780.23
Pubmed	54531	72215	107915	72.491	533.495	21.818	4083.08	4224.54
Yago	27038	39638	39181	44.495	229.416	12.256	2605.56	2622.23

Table 4: Comparison between 3HOP and Path-Tree

compression approaches, including the simple graph covering approaches, 2-hop, and 3-hop? We plan to investigate these problems in the future.

8. 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, 1989.

[2] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1–39, 2008.

[3] M.de Berg, M.van Kreveld, M.Overmars, and O.Schwarzkopf. *Computational Geometry*. Springer, 2000.

[4] Li Chen, Amarnath Gupta, and M. Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 493–504, 2005.

[5] Yangjun Chen and Yibin Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, pages 893–902, 2008.

[6] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, pages 961–979, 2006.

[7] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, pages 193–204, 2008.

[8] V. Chvátal. A greedy heuristic for the set-covering problem. *Math. Oper. Res*, 4:233–235, 1979.

[9] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the 13th annual ACM-SIAM Symposium on Discrete algorithms*, pages 937–946, 2002.

[10] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.*, 18(1):30–55, 1989.

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

[12] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD Conference*, pages 595–608, 2008.

[13] Richard Johnsonbaugh and Martin Kalin. A graph generation software package. In *SIGCSE '91: Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, pages 151–154, New York, NY, USA, 1991. ACM.

[14] Guy Kortsarz and David Peleg. Generating sparse 2-spanners. In *SWAT '92: Proceedings of the Third Scandinavian Workshop on Algorithm Theory*, pages 73–82, 1992.

[15] R. Schenkel, A. Theobald, and G. Weikum. HOPI: An efficient connection index for complex XML document collections. In *EDBT*, 2004.

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

[17] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *SIGMOD Conference*, 2008.

[18] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 845–856, 2007.

[19] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 75, 2006.

[20] Xifeng Yan, Philip S. Yu, and Jiawei Han. Substructure similarity search in graph databases. In *SIGMOD Conference*, pages 766–777, 2005.