SCARAB: Scaling Reachability Computation on Large Graphs [*]

Ruoming Jin[†] Ning Ruan[†]
[†] Department of Computer Science
Kent State University
Kent, Ohio, USA
{jin,nruan,sdey}@cs.kent.edu

Saikat Dey[†] Jeffrey Yu Xu[‡]
[‡] Department of Systems Engineering &
Engineering Management
Chinese University of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

ABSTRACT

Most of the existing reachability indices perform well on small- to medium- size graphs, but reach a scalability bottleneck around one million vertices/edges. As graphs become increasingly large, scalability is quickly becoming the major research challenge for the reachability computation today. Can we construct indices which scale to graphs with tens of millions of vertices and edges? Can the existing reachability indices which perform well on moderate-size graphs be scaled to very large graphs? In this paper, we propose **SCARAB** (standing for SCAlable ReachABility), a unified reachability computation framework: it not only can scale the existing state-of-the-art reachability indices, which otherwise could only be constructed and work on moderate size graphs, but also can help speed up the online query answering approaches. Our experimental results demonstrate that SCARAB can perform on graphs with millions of vertices/edges and is also much faster then GRAIL, the state-of-the-art scalability index approach.

Categories and Subject Descriptors

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

General Terms

Performance

Keywords

Scalable reachability, Reachability backbone, Reachability join test

1. INTRODUCTION

Reachability is a fundamental operator on directed graphs. It answers whether a vertex u can reach another vertex v using a simple path ($?u \rightarrow v$). Computing reachability has been studied in a wide range of computer science disciplines, including software engineering, programming languages, and distributed computing. Early work on reachability in the database field dates back to its application to the recursion operator and knowledge management [1]. The recent emergence of rich graph data (from biology, social networks, software analysis, semantic web) poses new challenges for reachability computation and reignites interest in discovering good reachability indices [24, 22].

In the last several years, quite a few graph indexing approaches [5, 23, 8, 21, 16, 9, 6, 15, 25, 24, 4, 22, 14, 7] have been proposed to speed up answering reachability queries in database systems. All these approaches lie between two extreme reachability computation schemes, namely, online DFS/BFS and the complete transitive closure, and aim to balance between query time, index size, and construction cost. However, almost all of them face the *scalability* bottleneck for handling massive graphs, which are quickly arising from social networks (such as Twitter and WeiBo), the semantic web, and large domain ontologies (such as in the biomedical field). The majority of these approaches can only handle moderate size graphs having tens or hundreds of thousands vertices/edges; only a few barely reach the "million-vertices" threshold [9, 22, 14]. Though online search methods, such as DFS/BFS, can always perform on any size graphs, their query answering time grows linearly with graph size, too costly for very large graphs.

To deal with the scalability problem, Yildirim *et al.* recently proposed GRAIL, which is a refined DFS utilizing auxiliary interval labeling to prune the search space [24]. However, its overall reachability computation speedup compared with DFS is quite limited (comparable or even slower than DFS for many cases). Furthermore, though it tends to reject a "negative query" rather fast (when a vertex cannot reach another vertex) [24], its performance for confirming a "positive query" is still a major issue as it has to discover an actual path between the queried vertices. Also, GRAIL can be one or two orders of magnitude slower for answering random queries, even more for positive queries.

To sum, scalability is quickly becoming the major research challenge for reachability computation today: Can we construct indices which scale to graphs with tens of millions of vertices and edges? Can the existing reachability indices which perform well on moderate-size graphs be scaled to very large graphs? In this paper, we provide positive answers to these questions. Specifically, we propose **SCARAB** (standing for SCAlable ReachABility), a unified reachability computation framework: it not only can scale the existing state-of-the-art reachability indices, which otherwise could only be constructed and work on moderate size graphs, but also can help speed up the online query answering approaches. In the following, before we proceed to the introduction of SCARAB, we first review the existing reachability indexing methods and discuss the underlying reason for their scalability bottleneck.

[*] R. Jin and N. Ruan were partially supported by NSF CAREER award IIS-0953950 and J. Y. Xu was supported by Council of the Hong Kong grants 419008 and 419109.

SIGMOD'12, May 20–24, 2012, Scottsdale, Arizona, USA.

1.1 Prior Work on Reachability Computation and Scalability Bottleneck

To answer the reachability query in a directed graph, we can always transform it into a directed acyclic graph (DAG) by coalescing strongly connected components into vertices and answering queries on the DAG. Since a DAG is often much smaller than the original directed graph, it is the target for reachability indexing. Let $G = (V, E)$ be the DAG for a reachability query, with number of vertices $n = |V|$ and number of edges $m = |E|$.

Numerous reachability computation approaches [1, 13, 18, 10, 20, 5, 23, 8, 21, 16, 9, 6, 15, 25, 24, 4, 22, 14, 7] have been proposed and can be largely classified into three categories: *transitive closure compression*, *hop labeling*, and *refined online search*.

Category I (Transitive Closure Compression): This category aims to directly compress the transitive closure TC and assign each vertex u a compressed reachable set $TC(u)$. To determine the reachability from vertex u to v, vertex v only needs to check against $TC(u)$. Representative methods include chain representation [13, 6], interval representation [1, 18],dual-labeling [23], path-tree [16], and bit-vector compression [22]. Using interval-representation as an example, in the reachable set of a vertex u, any contiguous vertex segment is compressed to its start vertex and end vertex. For instance, if the complete transitive closure of u is $\{0, 1, 2, 3, 7, 8, 9\}$, it can be compressed into two intervals: $[0, 3]$ and $[7, 9]$. The seminal work [1] shows how to find an optimal tree for such a representation. The latest work [22] shows that the bit-vector compression methods, such as PWAH (Partitioned Word Aligned Hybrid compression scheme), can also significantly compress these contiguous vertex segments (considering the corresponding binary vector representation of a reachable vertex set) . Good surveys of these methods can be found in [14, 24].

This category of methods is generally faster than the methods in the other two categories. Indeed, on moderate size graphs, several independent studies have demonstrated that interval representation and path-tree are the best in terms of query answering time for reachability computation [14, 22, 24]. However, the basis of their success is also the very reason for their scalability bottleneck: even when the graph is sparse, as the number of vertices increases, so does the size of the materialized transitive closure, inevitably exceeding the main memory capacity. On a moderate 8-GB machine, the upper capability of most these techniques is around one million vertices. Though the compressed TC may be materialized and stored on disk, both its construction and its query performance can become prohibitively expensive due to the disk-access cost. To make things even worse, in order to produce the best compression, some of the techniques, such as tree-based interval representation [1], actually need to compute the complete TC first.

Category II (Hop Labeling)): The second category utilizes intermediary vertices to encode the reachability, i.e., each vertex records a list of intermediate vertices it can reach (L_{out}) and a list of intermediate vertices which can reach it (L_{in}). To answer the reachability query, a *join* process between the outgoing intermediate vertices of the start vertex and the incoming ones of the end vertex is performed to determine whether there is a common vertex (or one vertex in the first set can reach another in the second). Using two sets of labels, hop labeling may also be viewed as a *transitive closure factorization* [15]. Compared with the first category methods, the hop labeling approaches are generally slower but can produce smaller index size [24, 14].

The seminal 2-hop labeling approach proposed by Cohen *et al.* [10] is the first in this category; the recent 3-hop labeling by Jin *et al.* [15] utilizes a chain decomposition as the intermediary highway structure to improve the 2-hop labeling; and more recently, path-hop [4] further generalizes 3-hop by utilizing a tree structure to replace the chain decomposition. However, all these approaches have high construction cost, which directly results in their scalability bottleneck. Specifically, in order to minimize the labeling size, the original 2-hop relies on a greedy set-cover framework, which not only involves repetitively finding densest subgraphs from a set of bipartite graphs, but also needs to materialize the entire transitive closure. The overall construction complexity of the original 2-hop ($O(n^3|TC|)$) is prohibitively expensive. Even with significant reduction of the construction cost by [20, 15, 4], these approaches can only handles graphs with far fewer than a million vertices.

Several heuristic approaches have been proposed to reduce 2-hop construction time. Schenkel *et al.* propose the HOPI algorithm, which applies a divide-and-conquer strategy to compute 2-hop labeling [20]. Cheng *et al.* propose several methods, such as a geometric-based algorithm [8] and graph partition technique [9], to produce a 2-hop labeling. Though their algorithms significantly speed up the 2-hop construction time, without the set-cover framework, they do not produce any approximation bound of their labeling size. Moreover, their scalability is also constrained by the lack of any good scalable partition algorithm on very large directed graphs, which these methods [20, 9] rely on.

Category III (Refined Online Search): The third category of methods utilize online search to answer reachability queries; they employ auxiliary labeling information to aggressively prune the search space. Specifically, Label+SSPI [5] and GRIPP [21] both utilize a tree cover to speed up the DFS process. The state-of-the-art GRAIL [24] assigns each vertex multiple interval labels; each label is generated by *random* depth-first traversals. The corresponding interval generated from the same DFS traversal can determine whether one vertex is likely to reach another: if $I_v \not\subseteq I_u$ (the interval of v is not a subset of the interval of u), then vertex u cannot reach vertex v; however, when $I_v \subseteq I_u$, we cannot determine whether u can reach v. Thus, $I_v \subseteq I_u$ is a necessary but insufficient condition for determining reachability; and multiple intervals can increase the rejection probability. GRAIL [24] utilizes such a multi-interval labeling to prune the search space in the DFS process and has been shown to be the best online search method. It is also the only feasible scalable solution which can handle graphs with tens of millions of vertices/edges so far.

The advantage of this category is that they generally do not need any optimization process and no transitive closure is needed in the construction. Its construction time and index size are both quite small, and thus can be applied to any graphs without size limitation. However, it generally has the slowest query answering time as it leaves most work to the query stage. When the graph size becomes very large, their query performance may become too expensive to answer reachability queries. As we mentioned earlier, even the state-of-the-art GRAIL has some issues on query performance.

1.2 Overview of SCARAB

To meet the scalability challenge of reachability computation on very large graphs, we develop a novel **SCARAB** approach, which can not only scale any of the existing reachability indices (such as methods in category I and II), but also speed up the online search methods (such as DFS and methods in category III). The basic idea of SCARAB is rather simple:

1. (Reachability Backbone) For any large graph, SCARAB first *scales down* the original graph by extracting a " reachability backbone" which carries the major "reachability flow" information.

2. (Accessing Backbone) To answer reachability query (u, v), start vertex u accesses a list of local *outgoing backbone vertices* and end vertex v accesses a list of local *incoming backbone vertices*. Then

u (v) perform a forward (backward) local BFS in the original graph to access the reachability backbone.

3. (Reachability Join Test) Given the outgoing backbone vertex set and the incoming backbone vertex set, a "reachability join test" determines whether any outgoing vertex can reach an incoming vertex in the backbone. If yes, then u can reach v; otherwise, no. Any existing reachability computation methods can be applied to the reachability join operation on the backbone.

Interestingly, SCARAB can be employed recursively; or in other words, we may construct a hierarchical backbone structure. Since the single level reachability backbone is already very scalable (sufficient to handles graphs with millions of vertices) as we will show in the empirical study, we will not consider the hierarchical structure in this work. The reachability backbone is similar in spirit to the highway structure used in several state-of-the-art shortest path distance computation methods on road networks [3, 19]. However, how to construct and utilize such structure in the reachability computation has not been fully addressed. Several existing approaches [20, 9, 25] have considered applying a graph partition to extract a high level structure to assist reachability computation. Unfortunately, the graph partition problem itself is known to be hard (especially on directed graphs) and lacks good scalable solution.

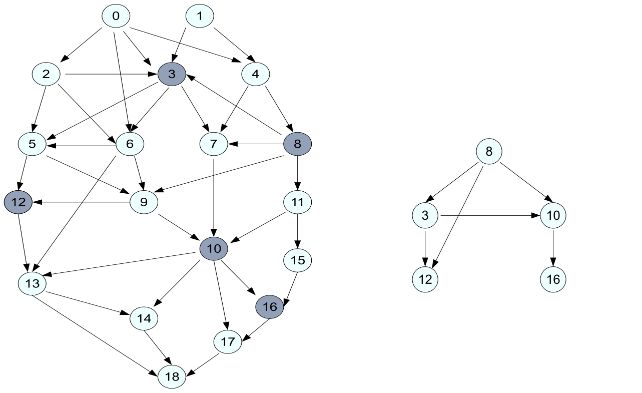
SCARAB needs to consider two basic research problems:

1. *How can we formally define the reachability backbone, and can it be discovered efficiently on very large graphs?* Here, the backbone itself not only needs to capture the high level reachability information of the original graphs, but also has to allow the fast access for any individual vertex.
2. *How can we utilize the reachability backbone to compute reachability efficiently?* Specifically: 1) *How can we access the backbone vertices quickly?* The local search cost must be minimized; 2) *How can we adopt and utilize the existing reachability index to optimize the reachability join test?* For different reachability computation methods, different strategies can be taken to speed up the reachability join test.

To answer the first question, we define the reachability backbone to be a *minimal* graph structure (in terms of the number of vertices), such that for every pair of vertices which are ϵ-hops apart in the original graph, both can access the backbone using only a local search (within ϵ-hops), and their corresponding access vertices are connected in the backbone. In other words, the backbone structure carries all non-local reachability information. To discover the backbone, we develop a set-cover approach which can approximate the minimal backbone with guaranteed bound and a fast heuristic approach which scales almost linearly with respect to the graph size. To speed up backbone access when answering reachability query, we consider the strategy to materialize those locally accessible backbone vertices for each vertex. For different categories of reachability computation, including online search, transitive closure compression, and hop-labeling, we tailor different strategies for faster reachability join test using the backbone.

2. REACHABILITY BACKBONE DEFINITION

In this section, we formally define the reachability backbone which plays a central role in SCARAB for scaling the reachability computation. Intuitively, it is designed to have a number of desired features: 1) it should be much smaller than the original graph; 2) it should carry sufficient topological information to assist the reachability computation in the original graph; 3) it should be easy to access for any vertex in the original graph. To satisfy these features, SCARAB explicitly separates *local* vertex pairs from *non-local* vertex pairs, and focuses on utilizing the backbone for recovering the reachability for non-local reachable pairs. For local pairs, reachability can be computed directly online, so no global information is needed. Furthermore, the separation between *local* and *non-local* vertex pairs is determined through a threshold parameter ϵ which can be used not only to facilitate the access of backbone vertices, but also to help control the backbone size.



(a) Original Graph G (b) G's Reachability Backbone

Figure 1: Running Example

Formally, given a *locality threshold* ϵ, for any pair of vertices u and v, if u can reach v within ϵ intermediate vertices, i.e., the distance between from u to v is no greater than ϵ, then (u, v) is referred to as a **local pair**, and if u can reach v but through at least $\epsilon + 1$ intermediate vertices, i.e., their distance is greater than ϵ, then (u, v) is referred to as a **non-local pair**. If u cannot reach v, then, (u, v) is referred to as a **unreachable** pair. Given this, the reachability backbone is defined as follows:

DEFINITION 1. (**Reachability Backbone**) *Given a DAG $G = (V, E)$ and the locality threshold ϵ, a reachability backbone $G^* = (V^*, E^*)$, where $V^* \subseteq V$ and E^* may contain edges not in E, has the following property: for every non-local (*unreachable*) pair (u, v) in graph G, there must (*not*) exist two vertices u^* and v^* in V^*, such that (u, u^*) and (v^*, v) are both local pairs in G and u^* can reach v^* in G^*.*

EXAMPLE 2.1. *Figure 1(b) shows a reachability backbone of graph G (Figure 1(a)) with $\epsilon = 2$. As an example, for non-local vertex pair $(1, 18)$, there is a backbone vertex 3 where vertex 1 reaches 3 in one hop, there is another backbone vertex 10 where vertex 10 reaches 18 in two hops, and vertex 3 reaches 10 in the reachability backbone. Indeed, for any non-local vertex pair in Figure 1(a), you can find their corresponding local backbone vertices and they are connected in the reachability backbone (Figure 1(b)). On the other hand, if two vertices cannot reach one another, no additional connection in the backbone will make them reachable from one to another. In other words, there are no false positives for reachability using the reachability backbone.*

Clearly, the reachability backbone depends on the locality threshold ϵ. As we will show in the empirical study Section 5, for any real and synthetic graphs, a reachability backbone with $\epsilon \leq 4$ can already significantly reduce the size of the original graph G by an order of magnitude. More surprisingly, for almost all the real graphs which are publicly available for reachability study [23, 9, 15, 24], the reachability backbone even for $\epsilon = 2$ can reduce the number of vertices by one to two orders of magnitude. The detailed study on the selection of the locality threshold ϵ is discussed in Section 5.

Reachability Backbone Edge Set: Given a DAG $G = (V, E)$ and its reachability backbone $G^* = (V^*, E^*)$, let $TC(V^*)$ be the transitive closure of G on V^*, i.e., $TC(V^*) = \{(u^*, v^*) \in V^* \times$

$V^*|u^* \rightarrow v^*$ in G}. Furthermore, let $TC^*(V^*)$ be the **transitive reduction** [2] of $TC(V^*)$, i.e., $TC^*(V^*)$ contains the smallest (and unique) edge set which preserve all reachability information between any two vertices in V^*. Given this, we make the following observation of the edge set E^* in the reachability backbone:

LEMMA 1. *(***Backbone Edge Set***) Given any reachability backbone $G^* = (V^*, E^*)$ for $G = (V, E)$, $E^* \subseteq TC(V^*)$. In other words, E^* does not introduce any additional reachability information beyond those between any two vertices of V^* in the original graph G. Furthermore, $\mathbf{G}^* = (V^*, TC^*(V^*))$ is also a reachability backbone of G, where $TC^*(V^*)$ is referred to as the* canonical *backbone edge set of the backbone vertex set V^*.*

Clearly, if any additional reachability is introduced, then there is will be false positives. This violates the backbone definition. The complete proof of Lemma 1 is omitted due to space limitation. Lemma 1 has the following important implication.

COROLLARY 1. *For any candidate reachability backbone graph $G^* = (V^*, E^*)$ in a given graph G, where $V^* \subseteq V$ and $E^* \subseteq TC(V^*)$, for any unreachable pair (u, v) in G, it will remain unreachable using G^*.*

This is because no additional reachability information is added in E^* besides those in the original graph G, i.e., $E^* \subseteq TC(V^*)$. Thus, we only need to focus on recovering the reachability for the non-local pairs in the original graph using the reachability backbone and do not have to deal with the non-reachable pairs. To facilitate our discussion, in the reminder of the paper, any *valid* backbone edge set E^* satisfies $TC^*(V^*) \subseteq E^* \subseteq TC(V^*)$.

EXAMPLE 2.2. *In Figure 1(b), the edge set of the reachability backbone is a valid backbone edge set for the backbone vertex set $\{3, 8, 10, 12, 16\}$. However, it is not a canonical backbone edge set. If we remove the redundant edges ($(8, 10)$ and $(8, 12)$), then the resulting edge set is a canonical one as any further edge removal will disconnect some reachability pair in the original graph.*

Minimal Reachability Backbone (MBR): Since the reachability backbone G^* aims to scale-down the original graph G, its size should be as small as possible while still maintaining its property for reachability computation. Given this, we introduce the **minimal reachability backbone** discovery problem: *given a DAG $G = (V, E)$ and the locality threshold ϵ, a minimal reachability backbone is the one with the smallest number of backbone vertices, i.e.,* $\arg\min_{|V^*|} G^*$. Since any reachability backbone edge set E^* satisfies $E^* \subseteq TC(V^*)$, then, we can first discover the backbone vertex set V^* on the graph G without defining its edge set E^*. Once the backbone vertex set is discovered, we can always choose $E^* = TC^*(V^*)$ as the default edge set, which can be immediately computed. Thus, the MRB problem can be reformulated as follows:

DEFINITION 2. *(***Minimal Reachability Backbone Vertex Set (MRBVS) Discovery***) Given a DAG $G = (V, E)$ and the locality threshold ϵ, we would like to find a* minimal *backbone vertex set $V^* \subseteq V$ such that for any non-local pair (u, v) in graph G, there must exist two vertices u^* and v^* in V^*, such that (u, u^*) and (v^*, v) are both local pairs in G and u^* can reach v^* in G.*

However, computing MRBVS is an NP-hard optimization problem because its corresponding decision problem is NP-hard.

THEOREM 1. *(***NP-hardness of MRBVS discovery problem***) Given a DAG $G = (V, E)$ and the locality threshold ϵ, computing its minimal backbone vertex set is NP-hard.*

The proof can be found in Appendix.

3. BACKBONE DISCOVERY

Since discovering the minimal backbone vertex set (MRBVS) is NP-hard, we cannot expect to find a exact solution in polynomial time. Furthermore, based on Definition 1, even the direct verification of whether a vertex subset in V meets the backbone criterion is computationally expensive: the reachability for any non-local pair and any unreachable pair has to be explicitly verified. In this section, we propose two backbone discovery algorithms to deal with the problem.

3.1 Backbone with Local Meeting Criterion

The first approach utilizes the *local meeting criterion* to find reachability backbone vertex set and then to discover MRBVS. Specifically, it is based on the following key observation:

LEMMA 2. *(***Local Meeting Criterion***) Given DAG $G = (V, E)$ and a subset of vertices V^*, if for any non-local vertex pair (u, v) with $d(u, v) = \epsilon + 1$, there exists a vertex $x \in V^*$, such that $u \rightarrow x, x \rightarrow v$ with $d(u, x) \leq \epsilon$ and $d(x, v) \leq \epsilon$, then V^* is a reachability backbone vertex set.*

Proof Sketch: Clearly, when $d(u, v) = \epsilon + 1$, the case is trivial and u^*=v^*=x. Now, let $d(u, v) > \epsilon + 1$. In that case, there is a vertex w such that $d(u, w) = \epsilon + 1$ and $w \rightarrow v$. Based on the postulate, we can find a vertex $x \in V^*$ such that $d(u, x) \leq \epsilon$ and $d(x, w) \leq \epsilon$. Let $u^* = x$. If $d(x, v) \leq \epsilon$, then $v^* = x$. Otherwise, we can find w', such that $d(w', v) = \epsilon + 1$ and $x \rightarrow w'$. Based on the postulate, we can find a vertex $y \in V^*$ such that $d(w', y) \leq \epsilon$ and $d(y, v) \leq \epsilon$. Then we have $v^* = y$. To sum, for any non-local pair (u, v), we can find u^* and v^* in V^*, such that $d(u, u^*) \leq \epsilon$, $d(v^*, v) \leq \epsilon$, and $u^* \rightarrow v^*$. □

Once a set of reachability backbone vertices V^*, which satisfy the local meeting criterion is discovered, generating its backbone edge set E^* is very easy: *for each vertex $u \in V^*$, add only edges in E^* linking u to only vertices in its ϵ-neighborhood.* The following lemma guarantees that the produced graph (V^*, E^*) maintains the reachability information in V^*, and can be used for recovering reachability between any non-local pair in the original graph.

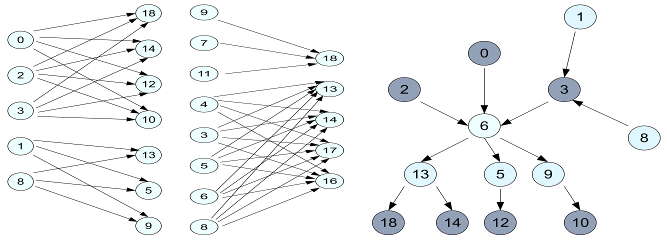
LEMMA 3. *(***Reachability Backbone Edge Set with Local Meeting Criterion***) Let V^* be the reachability backbone vertex set which satisfies the local meeting criterion in G and E^* contains the edges which directly link any local-pair in V^*, i.e., for any $(u, v) \in E^*$, $d(u, v) \leq \epsilon$ in G. Then if $u \rightarrow v$ in G ($u, v \in V^*$), then $u \rightarrow v$ in G^*=(V^*,E^*). In other words, $TC^*(V^*) \subseteq E^* \subseteq TC(V^*)$.*

EXAMPLE 3.1. *In Figure 1(a), the vertex set $\{3, 8, 10, 12, 16\}$ satisfies the local meeting criterion. Its corresponding edge set in Figure 1(b) is generated based on the above method.*

The proof of Lemma 3 is in the Appendix. Note that even though the local meeting criterion is very helpful in constructing a reachability backbone, not every reachability backbone vertex set has to satisfy the local meeting criterion.

EXAMPLE 3.2. *Consider graph G contains two sets of vertices A and B, and any vertex pair (a, b) ($a \in A$ and $b \in B$, and $d(a, b) = \epsilon + 1$), and these pairs are linked by vertex-disjoint paths (with length $\epsilon + 1$) and any two paths can only meet at the ends. Clearly, vertex set $A \cup B$ can be a reachability backbone vertex set (assuming there is no other vertices besides A, B and intermediate vertices in the paths linking these two sets).*

However, the local meeting criterion is much easier to manage and it also provides a good collection of possible backbone vertex sets. Especially, we observe the simple bound:



(a) C_6 (b) C_{10} (c) $N_\epsilon(6)$ and $N'_\epsilon(6)$.

Figure 2: Bipartite Graph Representation for Candidate Sets ($\epsilon = 2$) and its Generation

LEMMA 4. *Let V^*_ϵ be the minimal reachability backbone vertex set which satisfies the local meeting criterion with respect to the locality threshold ϵ and V^* be the overall minimal reachability backbone vertex set (not necessarily satisfying the local meeting criterion) with respect to the locality threshold ϵ, then, $|V^*_\epsilon| \geq |V^*| \geq |V^*_{\epsilon+1}|$.*

Proof Sketch: It is easy to verify that any reachability backbone vertex set (not necessarily satisfying the local meeting criterion) with locality threshold ϵ is always a reachability backbone vertex set which satisfies the local meeting criterion with respect to the locality threshold $\epsilon + 1$. Together with Lemma 2, the bound holds. □

Thus, V^*_ϵ provides an upper-bound of V^*. Formally, the problem of discovering the minimal reachability backbone vertex set with the local meeting criterion is referred to as the **LMRBVS discovery** problem and we will focus on this problem for minimal reachability backbone vertex discovery.

THEOREM 2. *(***NP-hardness of LMRBVS discovery problem***) Given a DAG $G = (V, E)$ and the locality threshold ϵ, computing its minimal backbone vertex set which satisfies the local meeting criterion is NP-hard.*

Theorem 2 can be proved similarly as the proof of Theorem 1 and is thus omitted for simplicity. Though the LMRBVS discovery problem is still NP-hard, it does admit an approximation algorithm based on the set-cover framework with guaranteed bound.

3.1.1 A Set-Cover Based Approach

Given this, we observe the LMRBVS discovery problem can be directly coded as an instance of the set cover problem [12]: Given DAG $G = (V, E)$ and the locality parameter ϵ, let $\mathcal{U} = \{(u, v)|d(u, v) = \epsilon + 1\}$ be the ground set, which includes all the non-local pairs with distance equal to $\epsilon + 1$. Each vertex x in the graph is associated with a set of vertex pairs $C_x = \{(u, v)|d(u, x) \leq \epsilon, d(x, v) \leq \epsilon, d(u, v) = \epsilon + 1\}$, where C_x includes all of the non-local pairs with distance equal to $\epsilon + 1$, such that u can reach x and x can reach v, each within ϵ hops. Thus, we have a total of $|V|$ candidate sets $\mathcal{C} = \{C_x | x \in V\}$. Now, in order to discover the LMRBVS, we seek the *a subset of vertices $V^* \subseteq V$, which has the minimal cardinality, to cover the ground set, i.e., $\mathcal{U} = \bigcup_{v \in V^*} C_v$*. Basically, V^* serves as the index for the selected candidate sets to cover the ground set.

EXAMPLE 3.3. *Figure 2 shows the candidate sets of vertex 6 and 10 for the graph in Figure 1(a). Here, each directed edge in the bipartite graph corresponds to a non-local pair with distance 3 for locality parameter $\epsilon = 2$.*

For this set cover instance, we may apply the classical greedy algorithm to find the minimal set cover, which essentially correspond to the LMRBVS: *Let R be the covered non-local pairs with distance $\epsilon + 1$ (initially, $R = \emptyset$). For each candidate set C_x in $\mathcal{C}$ (corresponding vertex x in V), we define the price of H as:*

$$\gamma(C_x) = \frac{1}{|C_x \backslash R|}.$$

At each iteration, the greedy algorithm picks up the candidate set C_x (vertex x)with the minimum $\gamma(H)$ (the cheapest price) and puts it into V^*. Then, the algorithm will update R accordingly, $R = R \cup C_x$. The process continues until no element in the ground set is uncovered: $R = \mathcal{U}$. It has been proven that the approximation ratio of this algorithm is $ln(|\mathcal{U}|) + 1$ [12].

Putting these together, we claim the following optimality result for discovering LMRBVS. Its proof is omitted for simplicity.

THEOREM 3. *The set-cover approach finds a reachability backbone vertex set with the local meeting criterion whose size is larger than the smallest cardinality of such a vertex set by at most $O(\ln(|\mathcal{U}|) = O(\ln n)$ factor where n is the number of vertices in the original graph G.*

Computational Complexity: The overall computational complexity of the set-cover approach is as follows. Let $N_\epsilon(v)$ and $E_\epsilon(v)$ denote the vertices and the edges, respectively, in v's forward ϵ-neighborhood. If directed edges are traversed in reserve, $N'_\epsilon(v)$ and $E'_\epsilon(v)$ are the vertices and edges of the reverse neighborhoods. First, we generate the ground set by performing a local BFS on each vertex u to discover all vertices which u can reach with $\epsilon + 1$ hops. This takes $O(\sum_{v \in V}(|N_{\epsilon+1}(v)| + |E_{\epsilon+1}(v))|)$. Second, to generate all candidate sets, for each vertex u, we perform two local BFS traversals, one forward and one backward on edges (with both stopping at depth ϵ). Figure 2 (c) shows the forward and reverse ($\epsilon = 2$)-neighborhood for vertex 6 in the running example graph (Figure 1(a)) and Figure 2 (a) is the resulting candidate set C_6. Then, any vertex pair $(x, y) \in N'_\epsilon(u) \times N_\epsilon(u)$, which belongs to ground set, i.e., their distance is $\epsilon + 1$, needs to be added to the candidate set C_u. This step takes $O(\sum_{v \in V}(|N_\epsilon(v)| + |E_\epsilon(v)| + |N'_\epsilon(v)| + |E'_\epsilon(v)| + |N_\epsilon(v)| \times |N'_\epsilon(v)|)$ time. Finally, the fastest set cover algorithm [11] can perform in linear time with respect to the size of candidate sets, i.e., $O(\sum_{v \in V} |C_v|)$, where $|C_v| \leq |N_\epsilon(v)| \times |N'_\epsilon(v)|$.

However, large scale-free graphs may contain some vertices with high out-degree and/or in-degree, which may produce very large ground set and candidate sets and make their materialization very costly. This can become the scaling bottleneck of this approach.

3.2 Fast and Scalable Backbone Discovery

Though the set cover approach can provide good approximation of MRBVS, it can be expensive for large graphs. Here, we describe a fast algorithm which need not materialize the ground set (and candidate sets) and which is very scalable, as each vertex needs to perform only a simple local BFS traversal (within ϵ hops). Instead of relying on the *local meeting criterion* which has the need for two BFS traversals (forward and reverse) and a Cartesian product between two sets, this approach utilizes a slightly different *one-side condition*. In particular, there is only one difference between the local meeting criterion and the one-side condition: the latter targets the *local* vertex pair with distance ϵ whereas the former targets the *non-local* vertex pair with distance $\epsilon + 1$.

Formally, given DAG $G = (V, E)$ and a subset of vertices V^**,for a vertex pair (u, v) in G with $d(u, v) = \epsilon$, if there is a vertex $x \in V^*$, such that $u \rightarrow x$ and $x \rightarrow v$, with $d(u, x) \leq \epsilon$ and $d(x, v) \leq \epsilon$, then we say (u, v) is* **covered** *by V^**. Otherwise, (u, v) is not covered by V^*.

LEMMA 5. *(**One-side Condition**) If V^* can cover every vertex pair (u, v) with $d(u, v) = \epsilon$ in G, then V^* is a reachability backbone vertex set.*

Proof Sketch: Based on the proof of Lemma 2, we just need to prove u can reach v using the backbone when $d(u, v) = \epsilon + 1$. Clearly, there exists a path with length $\epsilon + 1$, such as $x_0 = u, x_1, \cdots x_\epsilon, x_{\epsilon+1} = v$. We consider the following cases:
1) $u \in V^*$ and $v \in V^*$: Since both V^* $(u \rightarrow v)$ and we can always utilize the default edge $E^* = TC^*(V^*)$ in the reachability backbone, thus, V^* meets the criterion of reachability backbone;
2) $u \in V^*$ and $v \notin V^*$: For vertex pair (u, x_ϵ), there is $x \in V^*$ such that $d(u, x) \leq \epsilon$ and $d(x, x_\epsilon) \leq \epsilon$. Now, if $d(x, x_\epsilon) < \epsilon$, then, $d(x, v) \leq d(x, x_\epsilon) + d(x_\epsilon, v) \leq \epsilon$. If $d(x, x_\epsilon) = \epsilon$, then there is a direct neighbor of x, such that $d(x, y) = 1$ and $d(y, x_\epsilon) = \epsilon - 1$. Now, for vertex pair (y, v), we have $d(y, v) = \epsilon$. Thus, we must have $z \in V^*$, such that $d(y, z) \leq \epsilon$ and $d(z, v) \leq \epsilon$. Thus, we can find x and z in V^*, such that $x \rightarrow z$ and $d(u, x) \leq \epsilon$ and $d(z, v) \leq \epsilon$;
3) $u \notin V^*$ and $v \in V^*$ and 4) $u \notin V^*$ and $v \notin V^*$ can be proved similarly. Put together, we prove the lemma. □

Note that we also refer to the test condition for the reachability backbone in Lemma 5 as *one-side condition* based on the following property. For any vertex u, let $S_\epsilon(u)$ contain all the vertices which u reaches using exactly ϵ hops, i.e., $S_\epsilon(u) = \{v | d(u, v) = \epsilon\}$. *If $u \in V^*$, then any $(u, v) \in \{u\} \times S_\epsilon(u)$ is covered.* To facilitate our discussion, if any $(u, v) \in \{u\} \times S_\epsilon(u)$ satisfies the one-side condition, we say vertex u is *covered*, otherwise, it is *not covered*. Utilizing this property, we can utilize a fast heuristic approach to generate V^*.

Algorithm 1 FastCover(G)

```
sort vertices in $V$ based on certain order
$V^* \leftarrow \emptyset$
for each $u \in V$ do
    if NOT covered$(u, V^*)$ {$\{u\} \times S_\epsilon(u)$ is uncovered} then
        add $u$ to $V^*$
    end if
end for
return $V^*$;
Procedure covered$(u, V^*)$
$depth(u) \leftarrow 0$; $distance(v) \leftarrow \epsilon + 1$;
add $u$ to $Q$ {priority queue $Q$ ordered by topological order}
while $Q \neq \emptyset$ do
    $v \leftarrow Q.pop()$ {the one with least topological order};
    $v.visited \leftarrow TRUE$;
    $N(v) \leftarrow \{w : (w, v) \in E$ and $w.visited = TRUE\}$; {all
    end vertices of incoming edges of $v$}
    $depth(v) \leftarrow \min_{w \in N(v)} depth(v) + 1$;
    if $v \in V^*$ then
        $distance(v) \leftarrow 0$
    else
        $distance(v) \leftarrow \min_{w \in N(v)} distance(v) + 1$;
    end if
    if $depth(v) = \epsilon$ and $distance(v) \leq \epsilon$ then
        return FALSE; {$(u, v)$ is uncovered}
    end if
    add all $v$'s neighbors to $Q$;(if they are not in $Q$)
end while
return TRUE; {every vertex pair in $\{u\} \times S_\epsilon(u)$ is covered}
```

Algorithm 1 sketches the fast heuristic approach. We first order the vertices in certain way (determining the backbone vertex selection order). The most basic approach is to randomly order them (corresponding to an adaptive sampling procedure). Initially the reachability backbone vertex V^* is an empty set ($V^* = \emptyset$). Then, for each vertex u based on the order, we check whether it is *covered* by the current reachability backbone vertex set V^*, i.e., every vertex pair (u, v) where $v \in S_\epsilon(u)$ is covered. If it is not completely covered (*covered(u)=FALSE*), then, we simply add the vertex u into V^*. Based on the property of one-side condition, then we immediately cover u.

Given this, the major issue is that we need to quickly determine whether a vertex u is covered. The straightforward method needs to determine that every $(u, v) \in \{u\} \times S_\epsilon(u)$ is covered. This is clearly very expensive. Here, we describe a fast procedure which simply performs a single BFS of the neighborhood of u, i.e., $G[N_\epsilon(u)]$ which is the induced subgraph of all the vertices within ϵ hops of u including itself. Recall our goal is to check that for each vertex pair (u, v) with $d(u, v) = \epsilon$ to be covered in V^*, there exists a vertex $x \in V^*$, such that $u \rightarrow x$ and $x \rightarrow v$ with $d(u, x) \leq \epsilon$ and $d(x, v) \leq \epsilon$. We first make sure we will not visit any vertices which are more than ϵ hops away from u. This is easily done by recording the *depth* of each visited vertex. Furthermore, each vertex will record a variable *distance*, which records the smallest distance from an already visited backbone vertex x to it. The distance of each vertex is initialized to be $\epsilon + 1$, which suggests no backbone vertex reaches it in ϵ steps. For any visited vertex v in V^*, we assign $distance(v) = 0$. In particular, we visit the vertices based on their topological order, which has the property that for any visited vertex, all its predecessors must have been visited before. Given this, for any visited vertex v not in V^*, we simply choose the minimal distance of all its direct predecessors (the ends of incoming edges) and increase it by one. In this way, we can easily maintain the correct *distance* value for each vertex. This *covered* procedure is sketched in Algorithm 1.

The overall computational complexity of the *FastCover* procedure (assuming using random order) is $O(\sum_{u \in V} |N_\epsilon(u)| \log |N_\epsilon(u)| + |E_\epsilon(u)|)$. For random graphs with average vertex degree d, the time complexity can be written as $O(n\epsilon \log dd^\epsilon)$. Since this algorithm does not need to materialize the ground set and candidate sets, there is no scalability bottleneck and the algorithm scales linearly with respect to the graph size.

In addition, we note that there are many ordering strategies for determining the selection order of backbone vertices (besides the random ordering). Especially, we found the vertex order based on *the product of vertex in-degree and out-degree* is particularly effective for producing the reachability backbone on very large graphs. Though the *FastCover* does not provide any approximation bound, using this order strategy, this approach in most of the cases (in the empirical study) can discover the backbone vertex set with size being quite comparable to the set-cover approach (Section 5). Thus, we adopt this ordering strategy for *FastCover*. We note that such ordering will introduce an additional $O(|V| \log |V|)$ sorting time complexity. However, for real world large graphs, most of the products of vertex in-degree and out-degree are expected to be less than $O(|V|)$ due to the scale-free property. Given this, we may utilize the counting sort for the majority of vertices and thus empirically reduces the ordering cost to approximately $O(|V|)$.

Finally, it is also rather easy to construct the edge set for the reachability backbone vertex set with the one-side condition:

LEMMA 6. *Let V^* be the reachability backbone vertex set which satisfy the one-side meeting criterion in G and E^* contain the edges which directly link any local-pair in V^* and all the non-local pairs with distance $\epsilon + 1$ in V^*. Then if $u \rightarrow v$ in G $(u, v \in V^*)$, the $u \rightarrow v$ in G^*=(V^*, E^*).*

Note that the only difference between this lemma and Lemma 3 is that this one has to directly link non-local pairs with distance $\epsilon+1$, whereas Lemma 3 does not need. The reason can be observed in the first case in the proof of Lemma 5. Basically, there is possibility that two backbone vertices can be $\epsilon+1$ hops away and there is no other backbone vertices between them. The proof of Lemma 6 is similar to Lemma 3 and is omitted for simplicity.

4. REACHABILITY COMPUTATION VIA REACHABILITY BACKBONE

Using the reachability backbone G^* to compute a reachability query consists of two basic steps:

1. Local Search for Accessing Backbone: First, we perform two local BFS (within ϵ depth from the starting vertex) in the original DAG G: the forward BFS from u to collect $B^\epsilon_{out}(u)$, the set of all backbone vertices which u can reach within ϵ hops, i.e. $B^\epsilon_{out}(u) = \{x \in V^* | d(u,x) \leq \epsilon\} = V^* \cap N_\epsilon(u)$; and the reversed BFS from v to collect $B^\epsilon_{in}(v)$, the set of all backbone vertices which can reach v within ϵ hops, i.e., $B^\epsilon_{out}(v) = \{y \in V^* | d(y,v) \leq \epsilon\} = V^* \cap N'_\epsilon(v)$. Note that during the forward BFS (or reversed BFS), we can also check whether u reaches v locally ($d(u,v) \leq \epsilon$), and if it is, we directly confirm the reachability.

2. Reachability Join Test ($B^\epsilon_{out}(u) \rightarrow B^\epsilon_{in}(v)$): The reachability join test $B^\epsilon_{out}(u) \rightarrow B^\epsilon_{in}(v)$ in the reachability backbone G^* determines whether there exists $x \in B^\epsilon_{out}(u)$ and $y \in B^\epsilon_{in}(v)$, such that $x \rightarrow y$ in G^*. If it is then $u \rightarrow v$ in G and not otherwise. Given this, we basically need to compute the reachability between any $(x,y) \in B^\epsilon_{out}(u) \times B^\epsilon_{in}(v)$. Due to the modest size of the reachability backbone, we can effectively utilize any of the existing reachability indices and computational methods.

Algorithm 2 BasicReach(u,v)

```
Parameter: $G^*$ is the reachability backbone
perform two BFS to compute $B^\epsilon_{out}(u)$ and $B^\epsilon_{in}(v)$;
for each $x \in B^\epsilon_{out}(u)$ do
    for each $y \in B^\epsilon_{in}(t)$ do
        if Reach($x$,$y$|$G^*$) then
            return TRUE;
        end if
    end for
end for
return FALSE;
```

The basic reachability computation scheme is sketched in Algorithm 2. Note that $\mathbf{Reach}(x,y|G^*)$ is the generic reachability computation method (such as those describe in Section 1) in the reachability backbone G^*. In the following, we will discuss strategies and refinement to speed up the basic computation scheme. Specifically, in Subsection 4.1, we discuss optimization strategies to utilizing the transitive closure compression and hop-labeling approach. Then in Subsection 4.2, we describe how online search and GRAIL can be better adopted in this query scheme.

4.1 Speed Up Query Processing

In this subsection, we focus on optimization strategies for the reachability indexing approaches: the transitive closure compression (category I) and hop-labeling approach (category II) which are applied to the reachability backbone G^*. Specifically, for any method in the first category, each vertex x in the reachability backbone G^* is assigned a compressed transitive closure $TC(x)$ (different methods utilize different compression strategies) and to compute $\mathbf{Reach}(x,y|G^*)$, a search procedure quickly determines whether $y \in TC(x)$; for any method in the second category, each vertex x is assigned two labeling sets $L_{out}(x) \subseteq V^*$ (some vertices which x can reach) and $L_{in}(x) \subseteq V^*$ (some vertices which reach x). To compute $\mathbf{Reach}(x,y|G^*)$, $L_{out}(x)$ is searched against $L_{in}(y)$ to see whether there is a common vertex [10] (or there is a common chain [15, 4]) which can link x to y.

Given this, we can observe the performance of Algorithm 2 is determined by these two steps for any reachability query ($?u \rightarrow v$): 1) the two local BFS compute the backbone access vertex sets $B^\epsilon_{out}(u)$ and $B^\epsilon_{in}(v)$ and one of them is used to determine whether the start vertex can reach the end vertex locally. Here, the potential problem is that BFS may potentially scan a large number of vertices and edges, especially when there are hub vertices (incoming or outgoing) in the ϵ-neighborhood. 2) the cost of the reachability join test is determined by the number of reachability pair queries. In the basic scheme, we need to compute $|B^\epsilon_{out}(u)| \times |B^\epsilon_{in}(v)|$ pairs, which can be expensive.

Access Vertex Materialization and Reduction: To address these two problems, our first strategy is to explicitly materialize the backbone access vertex sets for each vertex u. This is because the number of those vertices is generally quite small. Interestingly, the actual materialized vertex set can be even smaller: given DAG G and its reachability backbone V^*, for each vertex u, the following two backbone access vertex sets need to be materialized:

$$\boldsymbol{\mathcal{B}}^\epsilon_{out}(u) = \{v \in V^* | d(u,v) \leq \epsilon \text{ and there is no other vertex } x, \\ \text{in } V^* \text{ such that } d(u,x) \leq \epsilon \wedge d(x,v) \leq \epsilon (u \rightarrow x \rightarrow v)\}$$

$$\boldsymbol{\mathcal{B}}^\epsilon_{in}(u) = \{v \in V^* | d(v,u) \leq \epsilon \text{ and there is no other vertex } y \text{ ,} \\ \text{in } V^* \text{ such that } d(v,y) \leq \epsilon \wedge d(y,u) \leq \epsilon (v \rightarrow y \rightarrow u)\}$$

LEMMA 7. (**Access Vertex Reduction**) *For any reachability query ($?u \rightarrow v$), when (u,v) is a non-local pair, it is sufficient to perform reachability join test between $\boldsymbol{\mathcal{B}}^\epsilon_{out}(u)$ and $\boldsymbol{\mathcal{B}}^\epsilon_{in}(v)$, i.e., $\boldsymbol{\mathcal{B}}^\epsilon_{out}(u) \rightarrow \boldsymbol{\mathcal{B}}^\epsilon_{in}(v)$, to determine whether u can reach v.*

Intuitively, Lemma 7 suggests that if a backbone vertex in V^* is accessed, then none of its successors (according to visit order) need to be consider in the reachability join test. This is because those pruned vertices are already in the backbone and can be accessed by those "first-accessed" ones. Therefore there is no need to record or utilize them in the reachability join test. Due to the space limitation, we omit the proof of Lemma 7. Since $|\boldsymbol{\mathcal{B}}^\epsilon_{out}(u)| \leq |B^\epsilon_{out}(u)|$ and $|\boldsymbol{\mathcal{B}}^\epsilon_{in}(v)| \leq |B^\epsilon_{in}(v)|$ for any vertex pair, this strategy can reduce the cost not only of online search but also of reachability join test.

Online Pruning: The second strategy targets directly the reachability join test. If we can quickly reject $x \rightarrow y$ where $x \in \boldsymbol{\mathcal{B}}^\epsilon_{out}(u)$ and $y \in \boldsymbol{\mathcal{B}}^\epsilon_{in}(v)$, then we do not need to actually perform $\mathbf{Reach}(x,y|G^*)$, which either involves searching through the compressed transitive closure of x, $TC(x)$ (in Category I) or comparing two labeling sets $L_{out}(x)$ and $L_{in}(y)$ (in Category II). Furthermore, if we can quickly reject $x \rightarrow v$, then we can even directly avoid the reachability tests against all vertices in $\boldsymbol{\mathcal{B}}^\epsilon_{in}(v)$.

To achieve such goal, we utilize the interval labeling method in GRAIL [24]. Basically, each vertex u in the entire graph G is assigned multiple interval labels $\mathcal{I}_u$ which can help to determine quickly the non-reachability between two vertices. These labels are generated by performing a constant number (c) of *random* depth-first traversals, i.e., the visiting order of the neighbors of each vertex is randomized in each traversal. Each traversal will produce one interval for every vertex in the graph. Such interval labeling has the property that if $\mathcal{I}_v \nsubseteq \mathcal{I}_u$, then vertex u cannot reach vertex v. However, when $\mathcal{I}_v \subseteq \mathcal{I}_u$, we cannot determine whether u can reach v. GRAIL [24] utilizes this labeling in the depth-first search to prune the search space. Such a labeling can be constructed very fast ($O(c(n+m))$) and its index size is only $O(cn)$, where c can be quite small ($c = 5$ is shown to be sufficient to provide good pruning) [24].

We use such labeling to help quickly reject any (x, y) pairs in $\boldsymbol{\mathcal{B}}^{\epsilon}_{out}(u) \times \boldsymbol{\mathcal{B}}^{\epsilon}_{in}(v)$ and any vertex x which cannot lead to $u \rightarrow v$. We explicitly compute each $\mathbf{Reach}(x, y|G^*)$ only if we cannot simply prune such a test using the multi-interval labeling. Note that for the hop-labeling approach (Category II), an alternative strategy exists which can directly avoid the explicit the pair-wise reachability computation. The idea is to first merge all the $L_{out}(x)$ for $x \in \boldsymbol{\mathcal{B}}^{\epsilon}_{out}(u)$ and $L_{in}(y)$ for all $y \in \boldsymbol{\mathcal{B}}^{\epsilon}_{in}(v)$, and then perform a comparison between the two merged lists. However, since the merge cost is actually quite expensive, we found this method is actually much slower than explicit pairwise comparison together with the online pruning method. Explicit pairwise comparison's early termination (when the first $x \longrightarrow y$ is confirmed) turns out to be quite effective. Thus, we do not adopt the merge strategy here.

Bidirectional Local Search: Though there is no need to perform the online BFS to collect the reachability backbone access vertices, we still need to determine whether u can reach v locally, i.e., $d(u, v) \leq \epsilon$. To perform such a local test, we can utilize a bidirectional BFS to reduce the search space. Specifically, the forward BFS starting at u needs to expand to at most $\lceil \epsilon/2 \rceil$ depth and the reversed BFS starting from v needs to expand to $\lfloor \epsilon/2 \rfloor$ depth. Furthermore, in either BFS expansion, if a reachability backbone vertex (in V^*) is visited, then we do not have to further expand its outgoing (or incoming) vertices, a considerable savings for hub vertices. Hub vertices (a vertex either with high in-degree of out-degree) tend to be covered in the reachability backbone vertex set. Indeed, if they are not covered, we can explicitly add them to the reachability backbone. Since the number of hub vertices tend to be quite small [17], this strategy can help reduce the cost of local search while not greatly expanding the backbone size.

Algorithm 3 FastReach(u,v)

```
Parameter: $G^*$ is the reachability backbone
Bidirectional online BFS search from $u$ and $v$;
if meet then
    return TRUE
end if;
for each $x \in \boldsymbol{\mathcal{B}}^{\epsilon}_{out}(u)$ do
    if $\mathcal{I}_v \subseteq \mathcal{I}_x$ then
        for each $y \in \boldsymbol{\mathcal{B}}^{\epsilon}_{in}(t)$ do
            if $\mathcal{I}_y \subseteq \mathcal{I}_x$ then
                if Reach($x$,$y$|$G^*$) then
                    return TRUE;
                end if
            end if
        end for
    end if
end for
return FALSE;
```

The query processing algorithm which incorporates the above optimization strategies is sketched in Algorithm 3. Clearly, its worst case computational complexity can be partitioned into two parts ($O(T_1 + T_2)$). T_1 comes from the bidirectional local search, where $T1 = \max_{u \in V}(|N_{\lceil \epsilon/2 \rceil}(u)| + |E_{\lceil \epsilon/2 \rceil}(u)|) +$ $\max_{v \in V}$ $(|N'_{\lfloor \epsilon/2 \rfloor}(v)| + |E'_{\lfloor \epsilon/2 \rfloor}|)$. T_2 is the cost of the reachability join test, given by $\max_{u,v \in V} |\boldsymbol{\mathcal{B}}^{\epsilon}_{out}(u)| \times |\boldsymbol{\mathcal{B}}^{\epsilon}_{in}(v)| \times T_3$ $\leq \max_{u,v \in V} |N_{\epsilon}(u)| \times |N_{\epsilon}(v)| \times T_3$, where T_3 is the worst case complexity of different reachability computational methods in the reachability backbone G^*. Recall that $|N'_{\epsilon}(v)|$ ($|E'_{\epsilon}(v)|$) is the number of vertices (edges) in v's reversed ϵ-neighborhood. For instance, consider Agrawal *et al.*'s tree-interval [1] is used to compress the transitive closure in the reachability backbone and let $n' = |V^*|$, and assume the original graph is a random DAG with average in-degree and out-degree d, then the worst case computational complexity of *FastReach* can be simplified to $O(d^{\lceil \epsilon/2 \rceil} + d^{2\epsilon}\dot{\log}n')$. As we will show in Section 5, the actual number of **Reach** invocations is much smaller than $d^{2\epsilon}$ and can be treated as constant (it is also a local measure). Thus, the worst case query computational complexity can be effectively scaled down and directly relates to the size of the reachability backbone.

4.2 Speed Up Online Search

The *FastReach* query processing scheme can be applied to the (refined) online search methods (Category III) such as GRAIL. Basically, each invocation of $\mathbf{Reach}(x, y|G^*)$ needs to perform an independent GRAIL search. However, this is clearly very expensive as each search needs to travel a large search space in G^* and the search spaces of different invocations can even overlap. Furthermore, assuming both y_1 and y_2 are in $\mathcal{B}^{\epsilon}_{in}(v)$, and $x \in \mathcal{B}^{\epsilon}_{out}(u)$, it may happen during $\mathbf{Reach}(x, y_1|G^*)$, it may reach y_2 even though x cannot reach y_1. Finally, for the online search method, the cost of local online search (for collecting access vertices in the reachability backbone) compared with the search in the reachability backbone is quite small. Thus, the need to actually materialize them is small.

Given this, *OnlineSearch* is proposed to deal with these issues and consists the following main steps:

1. *Perform a reversed BFS from* v *and for each visited reachability backbone vertex* $y \in V^*$*, flag it to be "target". If* u *is visited, return TRUE;*

2. *Perform a forward BFS from* u *and if any visited vertex* x *is a reachability backbone vertex* $x \in V^*$*, then perform a online search (recursive) from* u *in* G^**:*

2.1. *if the current visited vertex* x *is already visited before* ($visit[x] = TRUE$)*, then return (trace back);*

2.2. *if the current visited vertex* x *is a target* ($target[x] = TRUE$)*, then return TRUE;*

2.3. *recursively visited all* x*'s neighbors.*

3. *return FALSE;*

Basically, all the different searches starting from different backbone access vertices (in **2.**) can be considered as a single recursive graph traversal. To answer a reachability query (u, v), any vertex in G^* will be visited at most once. This is because we first flag all the backbone vertices which reach v within ϵ steps. Thus, if a vertex is already visited in the earlier search, then it basically has no chance to reach any of the flagged backbone vertices and no need to revisit them. Also, during the forward and reverse BFS, if a backbone vertex is visited, then there is no need to further explore it (similar to Lemma 7). In addition, we note for the refined online search, such as GRAIL, we can also utilize the interval labeling in both BFS in the original graph and recursive search in the reachability backbone search. For instance, in the reversed BFS, we only need to visit vertex y such that $\mathcal{I}_y \subseteq \mathcal{I}_u$, and in both forward BFS and online recursive search, we only need to visit vertex x such that $\mathcal{I}_v \subseteq \mathcal{I}_x$. Finally, we note that if we focus on the computational cost of the online recursive search as it is usually the dominant one, then the worst-case computational complexity of *OnlineSearch* is $O(n' + m')$, where $n' = |V^*|$ and $m' = |E^*|$ for any refined online search method [24, 5, 21].

5. EXPERIMENTAL EVALUATION

In this section, we empirically evaluate the **SCARAB** computation framework on both real and synthetic datasets. In particular, we are interested in following questions:

1. What are the effects of **SCARAB** on existing reachability indices, in terms of query time, index size, and construction time?

2. How do the state-of-the-art reachability computation methods scale when boosted by **SCARAB**? How do they compare with the state-of-the-art scalable reachability index, GRAIL?

3. How does the locality parameter ϵ affect the **SCARAB** reachability computation?

5.1 Experimental Setup

To answer these questions, we study the following state-of-the-art reachability approaches and their SCARAB counterparts.

1) *PathTree* [16], an improved version of Agrawal's tree-interval method [1];

2) *Nuutila's Interval* [18], a transitive closure compression method, recently demonstrated to be one of the fastest reachability computation methods [22];

3) *2HOP* [10], Cohen *et al.*'s original 2-hop labeling approach;

4) *GRAIL* [24], a scalable reachability indexing approach using random DFS labeling (the number of intervals is set at 5, as suggested by authors).

Here, Methods 1 & 2 are state-of-the-art transitive closure compression (Category I) methods, Method 3 is the optimal hop labeling approach (Category II), and Method 4 is the state-of-the-art online traversal method (Category III) and the best available scalability index. For each of these, we have developed their SCARAB methods, referred to as PT-G^*, IN-G^*, 2HOP-G^*, and GRAIL-G^*, where G^* refers to the reachability backbone. In addition, we also consider *DFS* (traditional depth-first search) and *PWAH* (the latest bit-vector compression method for transitive closure compression) [22] as benchmarks. For the latter, we use *PWAH-8*, the best variant of this approach [22]. All the methods (including source code) except DFS and 2HOP are either downloaded from authors' websites or provided by the authors directly. We coded DFS and 2HOP ourselves, with several fast heuristics [20, 15] for 2HOP to speed up its construction time (though it still cannot directly run on large graphs). The **SCARAB** framework utilizes the index construction and query processing implementation in the original methods. All algorithms are implemented in C++ based on the Standard Template Library (STL).

In the experiments, we focus on three important measures: query time, index size and construction time. For the query time, in past studies, only completely random queries are used, which have shown to be heavily skewed towards negative queries ($u \not\rightarrow v$) [24]. In some cases, there are even no any positive query ($u \rightarrow v$). This is highly unlikely for the real workload as the query pair tends to have certain connection. To address this issue, we also introduce the *equal* query load, where about 50% are positive queries and 50% are negative queries. Positive queries are generated by sampling the transitive closure. Since the **SCARAB** computation framework incorporates different indexing methods, our construction time consist of two parts: reachability backbone discovery time and indexing time. Correspondingly, the index size of a **SCARAB** method consists of three parts: the index size of the reachability backbone, the label size of GRAIL on the backbone, and the original graph size.

All experiments are performed on a Linux 2.6.18 machine with Intel Xeon 2.8GHz CPU and 12GB RAM.

5.2 Experimental Results

Here, we report on three groups of experiments to address the major questions on the **SCARAB** computation framework.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert V_S^*\rvert$	$\lvert E_S^*\rvert$	$\lvert V_F^*\rvert$	$\lvert E_F^*\rvert$
agrocyc	12684	13408	185	307	410	587
anthra	12499	13104	182	292	410	549
arXiv	6000	66707	1620	42789	3191	7684
citeseer	10720	44258	1300	4786	3951	8808
ecoo	12620	13350	190	317	409	589
go	6793	13361	1165	2499	1761	3182
kegg	3617	3908	185	187	487	495
mtbrv	9602	10245	175	306	407	557
nasa	5605	7735	938	1178	1010	1467
pubmed	9000	40028	642	3111	1899	8574
vchocyc	9491	10143	162	268	408	560
xmark	6080	7028	780	1308	744	1299
yago	6642	42392	190	697	1002	4057

Table 1: Small Real datasets

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert V_F^*\rvert$	$\lvert E_F^*\rvert$
citeseer	693,947	312,282	45,920	25,442
go_uniprot	6,967,956	34,770,235	137,055	1,436,198
mapped_100K	2,658,702	2,660,628	52,719	219,964
mapped_1M	9,387,448	9,440,404	184,856	825,950
uniprotenc_22m	1,595,444	1,595,442	31,909	31,908
uniprotenc_100m	16,087,295	16,087,293	467,047	467,046
uniprotenc_150m	25,037,600	25,037,598	1,046,951	1,046,950
citeseerx	6,540,399	15,011,259	336,670	1,531,727
cit-Patents	3,774,768	16,518,947	1,316,773	5,879,535

Table 2: Large Real datasets

Small Real Graphs (Studying SCARAB Effect): In the first group of experiments, we use a group of 13 small real graphs which have been used as the standard benchmarks in the recent studies on reachability index [23, 8, 16, 15, 25, 24, 4, 22, 14]. The first three columns in Table 1 show the dataset names along with the number of vertices and edges of their corresponding coalesced DAG. Table 3 reports the query times of DFS, GRAIL, 2HOP, PWAH-8, INTERVAL (IN), PATH-TREE (PT), and some of their **SCARAB** counterparts , including GRAIL-G^*, 2HOP-G^*, IN-G^* and PT-G^* using the *equal* query load. Table 4 reports the query time using the *random* query load. Specifically, we consider two types of reachability backbones: G_S^*, which is constructed by the set-cover approach (in Subsection 3.1) and the G_F^*, which is constructed by the *FastCover* approach (in Subsection 3.2). The locality parameter ϵ is set at 2. In Table 1, columns $|V_S^*|$ and $|E_E^*|$ record the number of vertices and edges of the reachability backbone G_S^* and columns $|V_F^*|$ and $|E_F^*|$ record the number of vertices and edges of G_F^*.

We make the following important observations on the query time:
1) For the original methods (no *SCARAB*), the methods of category I are clearly faster than the hop labeling approach (category II), which is faster than the online search methods (category III). PATH-TREE method consistently is the fastest at handling the equal query load; Nuutila's INTERVAL in some datasets is slightly faster than the PATH-TREE at handling the random query load. Overall, PATH-TREE is the clear winner. The latest bit-compression method (PAWH) can be one order of magnitude slower than both PATH-TREE and INTERVAL. In general, the reachability answering method slows down on the equal query load, though the effects on PATH-TREE and INTERVAL are quite minimal.

2) When the **SCARAB** framework is applied, GRAIL and 2HOP run 11 and 12 times faster, respectively. The reachability backbone discovered by the set-cover method (G_S^*) tends to provide better speedup than the ones discovered by the *FastCover* (G_F^*). This is consistent with the observation that the size of G_S^* is smaller than G_F^* (Table 1).

3) When applying the **SCARAB** framework to INTERVAL and PATH-TREE, there are moderate performance drop compared with the original ones though they are still consistently faster than the latest PAWH-8 method. Also, both backbones (G_S^*) and G_F^* have similar query performances. We note that the performance drop of

Dataset	Using Reachability Backbone								Original (Without Reachability Backbone)					
	GRAIL-G^*_S	GRAIL-G^*_F	2HOP-G^*_S	2HOP-G^*_F	IN-G^*_S	IN-G^*_F	PT-G^*_S	PT-G^*_F	DFS	GRAIL	2HOP	PWAH-8	INTERVAL	PATH-TREE
agrocyc	5.56	20.64	4.66	4.48	4.5	4.24	4.5	4.11	123.39	137.59	36.55	4.6	1.5	1.19
anthra	5.88	23.27	4.85	4.45	4.63	4.19	4.52	3.96	115.69	124.28	36.4	4.53	1.49	1.32
arXiv	89.02	153.28	32.64	24.29	19.31	13.51	20.87	14.97	4964.03	164.43	285.59	21.38	3.8	3.29
citeseer	44.62	48.59	21.73	17.04	17.42	14.2	16.62	14.55	414.41	101.65	273.22	55.84	5.32	3.61
ecoo	6.32	21.56	4.7	4.58	4.56	4.28	4.44	4.17	137.22	134.83	47.52	4.47	1.49	1.21
go	23.07	25.65	10.83	9.3	9.91	8.5	9.67	8.4	233.42	70.89	120.77	32.09	3.81	3.2
kegg	9.84	15.84	6.04	5.94	5.74	5.65	5.65	5.52	827.49	263.94	26.08	5.14	1.85	1.31
mtbrv	6.65	25.71	4.81	4.58	4.63	4.23	4.49	4.08	151.37	122.78	42.28	4.27	1.49	1.13
nasa	25.9	33.78	7.91	8.24	7.33	7.52	6.99	7.21	243.1	87.88	70	11.15	2.58	1.45
pubmed	45.04	51.28	24.66	15.2	16.17	12.23	17.02	13.7	382.55	136.24	513.49	73.64	5	4.13
vchocyc	6.2	24.95	4.58	4.39	4.47	4.11	4.35	4.06	139.23	120.38	44.53	4.53	1.48	1.09
xmark	148.41	162.54	10.01	10.31	9.33	9.58	8.4	8.69	351.99	118.14	83.97	40.06	2.78	1.52
yago	6.41	7.74	5.05	4.91	4.75	4.79	4.74	4.74	73.37	51.31	194.77	15.86	4.11	3.01

Table 3: Query Time (ms) Based on Equal Query of Real Datasets

Dataset	Using Reachability Backbone								Original (Without Reachability Backbone)					
	GRAIL-G^*_S	GRAIL-G^*_F	2HOP-G^*_S	2HOP-G^*_F	IN-G^*_S	IN-G^*_F	PT-G^*_S	PT-G^*_F	DFS	GRAIL	2HOP	PWAH-8	INTERVAL	PATH-TREE
agrocyc	2.64	2.66	2.6	2.63	2.67	2.67	2.63	2.6	23.69	138.38	25.29	1.57	1.25	1.35
anthra	2.57	2.6	2.53	2.55	2.61	2.59	2.59	2.55	21.4	124.16	25.17	1.39	1.23	1.36
arXiv	136.61	93.55	24.51	16.57	17.58	11.8	18.14	12.25	8163.23	165.82	130.5	30.46	5.79	4.36
citeseer	9.45	10.09	6.98	6.5	7.05	6.26	6.79	6.16	271.07	100.98	58.61	32.89	5.22	3.47
ecoo	2.62	2.62	2.58	2.58	2.66	2.63	2.62	2.57	27.75	134.63	25.32	1.46	1.25	1.3
go	3.78	3.94	2.95	2.87	2.97	2.91	2.92	2.84	78	70.4	60.78	5.69	3.29	2.44
kegg	6.99	11.42	5.2	5.17	5.18	5.05	5.07	5.01	1333.52	264.14	26.77	4.97	2.69	1.96
mtbrv	2.51	2.57	2.47	2.47	2.55	2.52	2.5	2.48	30.31	121.94	25.88	1.5	1.27	1.3
nasa	3.22	3.33	2.55	2.58	2.59	2.59	2.54	2.55	83.94	87.94	43.91	4.18	2.63	1.66
pubmed	8.31	9.52	6.5	5.88	6.26	5.59	6.27	5.78	234.15	136.35	63.68	37.11	4.12	3.34
vchocyc	2.49	2.56	2.47	2.45	2.53	2.51	2.5	2.46	30.83	120.67	26.86	1.49	1.26	1.34
xmark	8.85	9.27	3.75	3.8	3.76	3.8	3.72	3.72	246.4	118.68	48.2	6.69	2.61	1.77
yago	4.24	4.72	4.19	4.34	4.2	4.33	4.16	4.26	73.75	51.45	54.96	13.34	3.53	2.88

Table 4: Query Time (ms) Based on Random Query of Real Datasets

Dataset	Using Reachability Backbone				Original(WithoutBackbone)		
	GRAIL-G^*_F	IN-G^*_F	PT-G^*_F	2HOP-G^*_F	DFS	GRAIL	IN
citeseer	34.98	27.34	26.80	30.35	26.67	74.02	12.87
go_uniprot	68.74	26.30	25.07	27.56	191.90	116.73	26.74
mapped_100K	62.13	17.93	18.34	17.26	154.94	349.51	6.11
mapped_1M	42.86	19.38	23.55	20.07	131.79	405.41	6.47
uniprotenc_22m	16.69	17.08	16.39	16.47	22.73	62.04	16.32
uniprotenc_100m	34.89	30.60	37.51	722.85	34.77	112.54	21.29
uniprotenc_150m	55.40	81.30	—	—	40.63	117.05	37.07
citeseerx	1475.00	36.84	—	—	38183.80	2148.26	11.33
cit-Patents	528.31	109.58	—	—	7449.32	556.58	—

Table 5: Query Time(ms) on Equal Query of Large Real Graphs

Dataset	Using Reachability Backbone				Original(WithoutBackbone)		
	GRAIL-G^*_F	IN-G^*_F	PT-G^*_F	2HOP-G^*_F	DFS	GRAIL	IN
citeseer	10.17	9.81	9.72	9.67	22.39	74.06	11.38
go_uniprot	18.97	18.13	16.13	17.17	252.83	92.85	20.58
mapped_100K	10.14	9.91	9.42	9.42	11.56	396.57	8.78
mapped_1M	11.89	12.32	12.47	13.20	12.07	406.03	6.93
uniprotenc_22m	15.00	14.47	14.55	13.65	26.24	83.33	16.21
uniprotenc_100m	23.77	22.69	21.26	23.91	39.99	96.43	25.46
uniprotenc_150m	46.40	75.09	—	—	43.97	124.80	32.18
citeseerx	114.83	34.16	—	—	74121.40	2148.33	16.60
cit-Patents	435.32	78.68	—	—	15888.80	691.70	—

Table 6: Query Time(ms) on Random Query of Large Real Graphs

using the reachability backbone mainly comes from the fact that *SCARAB* has to perform an additional local search. 4) In the random query load, we note that most **SCARAB** approaches have similar performance. This is because the online pruning method (Subsection 4.1 and 4.2) based on the GRAIL labeling is in general quite effective in rejecting negative queries. To sum, the **SCARAB** approach only moderately increases the query time of INTERVAL and PATH-TREE, but significantly speeds up GRAIL and 2HOP on these real datasets. Also, the query performance based on the *FastCover* is generally slower but still comparable to the one based on the *SetCover* approach. However, the former is much easier to construct. Due to space limitation, we will focus on studying the reachability backbone based on *FastCover* in the rest of the experimental evaluation.

Figure 3 shows the index size of different reachability index methods along with their **SCARAB** counterparts. Interestingly, the index size of the **SCARAB** variants is actually larger than the original ones without the reachability backbone. This is because all of them utilize the GRAIL labeling, which on these small graphs tends to be larger than other approaches. It is not hard to see in the **SCARAB** approaches, the other costs, including the materialized backbone access vertices, are quite small. Basically, all of their sizes are quite close to the index size of GRAIL. Figure 5 shows the construction time of different reachability indices. It is interesting to observe that the **SCARAB** approach generally has smaller construction time compared with their corresponding original ones. Basically, the very light preprocessing of *FastCover* can scale down the graph significantly. Basically, the very light preprocessing of *FastCover* can scale down the graph significantly. Indeed, in Table 1, even for $\epsilon = 2$, almost all the graphs can be scaled down by more than an order of magnitude. Thus, the original reachability index needs to be performed on a much smaller graphs (backbones).

Large Real Graphs (Studying SCARAB Scalability):

In this group of experiments, we study the scalability of the **SCARAB** framework. We use 9 large real graphs used in the GRAIL study [24], which Yildirim *et al.* show they cannot be processed by existing indexing methods, except for DFS and GRAIL. For the original methods without reachability backbone, we confirm such observation except the latest implementation of Nuutila's INTERVAL [22], which is published after the GRAIL paper. However, it also fails on one large dataset. Given this, we are interested in how **SCARAB** can scale the existing reachability indices which cannot process these large graphs. Here, we use $\epsilon = 2$ for generating backbones. Table 2 shows the size of the original DAG and its corresponding reachability backbone size. The average reduction rate is around 25 times.

Tables 5 and 6 report the query time using the *equal* and *random* query load, respectively. We make the following observations:
1) Similar to the results on small graphs, **SCARAB** provides a nice speedup over the original GRAIL. In several graphs, the speedup is more than an order of magnitude.
2) INTERVAL turns out to be quite scalable. But INTERVAL-G^* is quite comparable and even in some cases, slightly faster. Furthermore, INTERVAL-G^* handles the *cit-patents* graph easily with

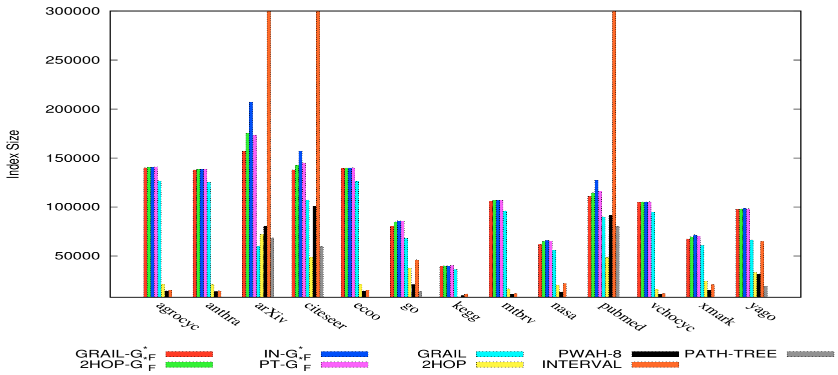


Figure 3: Index Size on Real Small Graphs

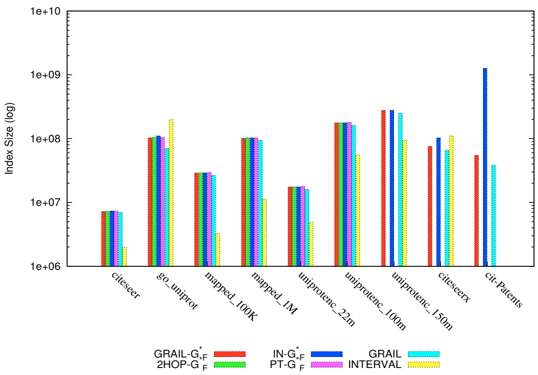


Figure 4: Index Size on Real Large Graphs

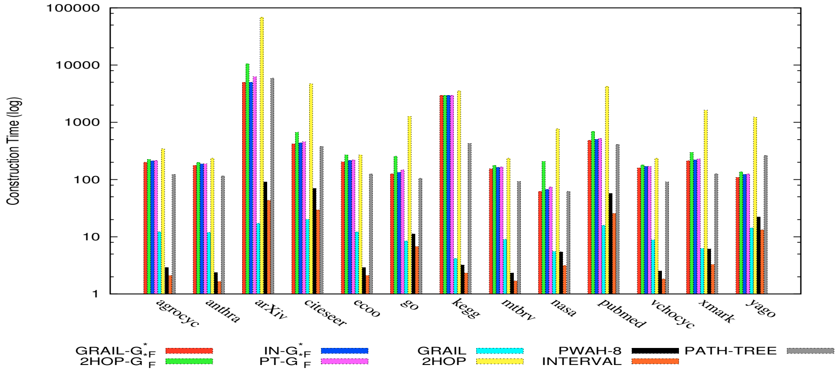


Figure 5: Construction Time (ms) on Real Small Graphs

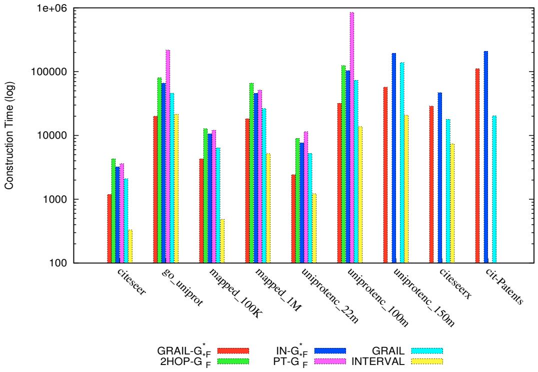


Figure 6: Construction Time (ms) on Real Large Graphs

very good query performance whereas INTERVAL cannot.
3) With the help of **SCARAB**, 2HOP and PATH-TREE using the reachability backbone can handle 6 out of 9 large graphs, which they had difficulty handling before. The largest one contains more than 16 million vertices.
Figure 4 and Figure 6 show the index size and the construction time of different reachability indices, respectively. Similar observations as in first group of experiments can be made. Interestingly, in several large datasets, the index size of the **SCARAB** variants is still dominated by the GRAIL labeling.
Large Scale-Free Graphs (ϵ effects on G^*): In the last group of experiment, we study the effect of ϵ on the size of reachability backbone G^* produced by the *FastCover* method. Here, we vary the ϵ from 2 to 4 on a group of large scale-free directed graphs. The number of vertices in these graphs range from $1,000,000$ vertices to $10,000,000$ vertices. The edge density of these larges is fixed to be 2.

Table 7 shows the number of vertices and edges in G^* with respect to ϵ on these large graphs. Clearly, as ϵ increases, the size of backbone also reduces. However, the density of the backbone will also increase accordingly. This is expected as the local search range for each backbone vertex increases. But we note that the increase rate is rather small. Interestingly, we note that even when $\epsilon = 2$, the vertex reduction rate is around 4 to 5 times. Also, this seems to be lower than the real graphs. This suggests that there are still some important properties of the real world graphs, which are not well captured by the existing graph model, such as the scale-free (power-law degree distribution) model. Finally, we note that the INTERVAL approach can handle the graph up to $3,000,000$ vertices. Using the **SCARAB** framework, it can handle all these graphs, and its query performance can be more than one order of magnitude faster than GRAIL. Duo to the space limitation, we do not report the query time, index size, and construction time on these graphs.

Dataset	$\epsilon = 2$		$\epsilon = 3$		$\epsilon = 4$	
	$\lvert V_F^*\rvert$	$\lvert E_F^*\rvert$	$\lvert V_F^*\rvert$	$\lvert E_F^*\rvert$	$\lvert V_F^*\rvert$	$\lvert E_F^*\rvert$
SF_1M	272951	948147	191208	828203	148411	786808
SF_2M	550860	1945895	385357	1698330	298336	1613006
SF_3M	823203	3037757	577754	2641205	448956	2499018
SF_4M	1125023	4020338	785084	3490157	605751	3303259
SF_5M	1335414	4532958	907895	3930738	698639	3768870
SF_6M	1597084	5509326	1077552	4766918	824395	4567924
SF_8M	2130229	7348966	1437845	6413161	1100027	6199982
SF_10M	2663122	9191144	1798266	8025036	1376667	7760405

Table 7: Size of Reachability Backbone on Scale-Free Graphs

6. CONCLUSION

In this paper, we introduce a simple yet effective **SCARAB** framework to help the existing reachability indices to handle large graphs. It can also help speed up the online query answering approaches. Our experimental results demonstrate that SCARAB can perform on graphs with millions of vertices/edges and is also much faster then GRAIL, the state-of-the-art scalability index approach. In the future work, we will investigate to apply **SCARAB** hierarchically or recursively. We also plan to study how **SCARAB** can be applied to other reachability problems, such as label-constraint reachability problems. Finally, we plan to study how to incrementally maintain the reachability backbone for the dynamic graphs.

7. REFERENCES

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

[2] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[3] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316:566–, April 2007.

[4] J. Cai and C. K. Poon. Path-hop: efficiently indexing large graphs for reachability queries. In *CIKM '10*, 2010.

[5] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB '05*, pages 493–504, 2005.

[6] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, 2008.

[7] Y. Chen and Y. Chen. Decomposing dags into spanning trees: A new way to compress transitive closures. In *ICDE'11*, 2011.

[8] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, 2006.

[9] 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 *EDBT*, 2008.

[10] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[11] G. Cormode, H. Karloff, and A. Wirth. Set cover algorithms for very large datasets. In *CIKM '10*, pages 479–488, 2010.

[12] U. Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

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

[14] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *TODS*, 36(1), 2011.

[15] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD'09*, 2009.

[16] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD'08*, 2008.

[17] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[18] E. Nuutila. *Efficient Transitive Closure Computation in Large Digraphs*. PhD thesis, Finnish Academy of Technology, 1995.

[19] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *17th Eur. Symp. Algorithms (ESA)*, 2005.

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

[21] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD '07*, 2007.

[22] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD '11*, pages 913–924, 2011.

[23] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE '06*, page 75, 2006.

[24] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, pages 276–284, 2010.

[25] L. Zhu, B. Choi, B. He, J. X. Yu, and W. K. Ng. A uniform framework for ad-hoc indexes to answer reachability queries on large graphs. In *DASFAA '09*, 2009.

APPENDIX

A. PROOF OF THEOREM 1.

Proof Sketch: We reduce the classical NP-hard problem, set-cover problem (SCP), to this problem In SCP, let $\mathcal{U}$ be the ground set and $\mathcal{C}$ records all the candidate sets, where for any candidate set $C \in \mathcal{C}$ and $C \subseteq \mathcal{U}$. The goal is to determine whether there are K (or less) candidate sets in $\mathcal{C}$ such that $\cup_i C_i = \mathcal{U}$. Now we transform it to the decision version of MRBVS discovery problem, i.e., whether there is a backbone vertex set containing K or less vertices.

We construct the following DAG based on a set cover instance: let $G = (X \cup Y \cup Z, E_{XY} \cup E_{YZ})$ be the DAG, where each vertex in X and Z corresponds to a unique element in the ground set $\mathcal{U}$, and each element in Y corresponds to a candidate set in $\mathcal{C}$; the edge set E_{XY} contains all the edges (x_u, y_C) where x_u is the corresponding vertices of element $u \in \mathcal{U}$ in the vertex set X, and y_C is the corresponding vertex of candidate set $C \in \mathcal{C}$ in Y, and the element $u \in \mathcal{U}$ belongs to the candidate set C in $\mathcal{C}$; the edge set $E_{YZ} = Y \times Z$, i.e., it contains the edge set which connects any pair from a vertex in Y to a vertex in Z.

Given this, we will show that for the locality parameter $\epsilon = 1$, the minimal reachability backbone vertex set (MRBVS) contains only vertices in Y and directly corresponds to the minimal set cover solution. Thus, the set cover problem which asks whether there is a K or less vertex cover can be directly answered by the solution on whether there is a K or less backbone vertex set in the above problem instance.

For the first claim (MRBVS contains only vertices in Y), if it is not, let the minimal reachability backbone vertex set $V^* = X' \cup Y' \cup Z'$, where X',Y', and Z' contains the backbone vertices from vertex set X, Y, and Z, respectively.

Case 1 ($X' \neq \emptyset$ and $Z' = \emptyset$): In this case, we can simply drop all the vertices in X', and $V^* = Z'$ is enough to serve as the backbone vertices. This is because for any non-local pair (x, z), there exist x^* and y^* where y^* must be in Y', and either (1) $x^* = x$ in X' or (2) $x^* = y^*$ in Y'. For (1), we must have $(x, y^*) \in E_{XY}$ and thus we can always replace x^* with y^*. Basically, Y' would contain enough backbone vertices. A similar proof holds for **$X' = \emptyset$ and $Z' \neq \emptyset$.**

Case 2 ($X' \neq \emptyset$ and $Z' \neq \emptyset$): In this case, we construct Y_s as follows: each vertex $x \in X'$ randomly chooses an edge (x, y) in E_{XY} and adds (x, y) into Y_s. It is easy to see that $|Y_s| \leq |X'$. We claim the following backbone vertex set $Y' \cup Y_s$ is enough to recover the reachability between any non-local pair in G. This is because for any non-local pair (x, z), which needs $x^* = x$ and $z^* = z$ in the original backbone vertex set, it now can use $y \in Y_s$ ($(x, y) \in E_{XY}$) to serve as the backbone vertices: $x^* = y$ and $z^* = y$. Since $|Y_s \cup Y'| < |X' \cup Y' \cup Z'|$, this is clearly impossible due to the minimality assumption. To sum, the minimal reachability backbone vertex set V^* should contain only the vertices in Y.

Now, we show that V^* directly corresponds to the solution of the minimal SCP. By way of contradiction, if the corresponding set of candidates in V^* is not the minimal one which can cover the ground set $\mathcal{U}$, and then we claim V' which corresponds to the minimal set cover is also a backbone vertex set. This is because for any non-local pair (x, z), there must exist y in V' where $(x, y) \in E_{XY}$ since x in the ground set is covered by some candidate C (corresponding to y). Clearly, this contradicts our assumption that V^* is the minimal one. □

B. PROOF OF LEMMA 3.

Proof Sketch: Since $(u, v) \in E^*$ only if (u, v) is local pair in G ($u \to v$), therefore, $E^* \subseteq TC(V^*)$. Now, we show $TC^*(V^*) \subseteq E^*$, i.e., if $u \to v$ in G, the $u \to v$ in G^*=(V^*,E^*). If $d(u, v) \leq \epsilon$, then $u \to v$ in G^*=(V^*,E^*) by definition. If $d(u, v) = \epsilon + 1$, then based on Lemma 2, there exists a $x \in V^*$ such that $d(u, x) \leq \epsilon$ and $d(x, v) \leq \epsilon$. So, based on the definition of G^*, $u \to v$. Now, if $d(u, v) > \epsilon + 1$, then there is a vertex w such that $d(u, w) = \epsilon + 1$ and $w \to v$. Based on Lemma 2, we can find a vertex $x \in V^*$ such that $d(u, x) \leq \epsilon$ and $d(x, w) \leq \epsilon$. So, the proof is reduced to $d(x, v)$. We apply the same procedure until we reach a vertex $y \in V^*$ with $d(y, v) \leq \epsilon$. Then, based on the definition of G^*, $u \to v$. □