The VLDB Journal
DOI 10.1007/s00778-013-0346-6

SPECIAL ISSUE PAPER

Efficient processing of k-hop reachability queries

James Cheng · Zechao Shang · Hong Cheng · Haixun Wang · Jeffrey Xu Yu

Received: 15 January 2013 / Revised: 21 October 2013 / Accepted: 19 November 2013

Abstract We study the problem of answering ***k*-hop reachability queries** in a directed graph, i.e., whether there exists a directed path of length k, from a source query vertex to a target query vertex in the input graph. The problem of k-hop reachability is a general problem of the classic reachability (where $k = \infty$). Existing indexes for processing classic reachability queries, as well as for processing shortest path distance queries, are not applicable or not efficient for processing k-hop reachability queries. We propose an efficient index for processing k-hop reachability queries. Our experimental results on a wide range of real datasets show that our method is efficient and scalable in terms of both index construction and query processing.

Keywords Graph indexing · k-hop reachability · Reachability index · Shortest path index · Distance queries

J. Cheng (✉)
Department of Computer Science and Engineering, The Chinese University of Hong Kong, New Territories, Hong Kong
e-mail: jcheng@cse.cuhk.edu.hk

Z. Shang · H. Cheng · J. X. Yu
Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, New Territories, Hong Kong
e-mail: zcshang@se.cuhk.edu.hk

H. Cheng
e-mail: hcheng@se.cuhk.edu.hk

J. X. Yu
e-mail: yu@se.cuhk.edu.hk

H. Wang
Microsoft Research Asia, Beijing, China
e-mail: haixunw@microsoft.com

1 Introduction

The **reachability query**, which asks whether one vertex can reach another vertex in a directed graph, is a basic operator for a variety of databases (e.g., XML, RDF) and network applications (e.g., social and biological networks). The problem of how to efficiently answer reachability queries has attracted a lot of interest lately [2,5–8,13–16,22,26–28,30,32–34,37,38]. There are also several extensions to the classic reachability problem; for example, reachability in uncertain graphs where the existence of an edge is given by a probability [24], and reachability with constraints such as edges on the path must have certain labels [23], etc.

In this work, we study a new type of reachability queries. Instead of asking whether a vertex t is reachable from a vertex s, we ask whether t is reachable within k hops from s. In other words, *the query asks whether there exists a path from s to t such that the length of the path is no more than k*. We call this problem the ***k*-hop reachability** problem.

The primary motivation for the k-hop reachability problem is that *in many real-life networks (e.g., wireless or sensor networks, the Web and Internet, telecommunication networks, social networks, etc.), the number of hops within which s can reach t indicates the level of influence s has over t*. Applications on such networks can benefit more from k-hop reachability than classic reachability (i.e., $k = \infty$). We give some examples as follows.

In a wireless or sensor network, where a broadcasted message may get lost during any hop, the probability of reception degrades exponentially over multiple hops [31]. In these applications, reachability may not be meaningful or have much practical use, while k-hop reachability, since it can model the level and sphere of the influence, is helpful in many analytical tasks.

Published online: 11 December 2013

In many real-life networks, the k-hop reachability between two vertices is of more interest when the value of k is small. For example, in social networks, although there is a well-known six-degrees-of-separation theory (i.e., any two persons are only 6 or fewer hops away from each other), the degree of acquaintance may even decrease super exponentially (i.e., two persons may hardly know each other if they are just 3 hops apart).

Clearly, we are more interested to know whether two persons are connected within only a few hops, instead of beyond 6 hops for which case one can almost know for sure they are connected. On the other hand, a small k does not necessarily make the problem easier. Consider an application that, given two persons, asks whether one is reachable from the other within 6 hops. A naive implementation is to invoke a breadth-first search (BFS). However, among the search quests in reality, a majority of them have at least one of the persons as a celebrity at some level of the BFS. A BFS from a celebrity (e.g., Lady Gaga, who has 40,000,000 fans on Facebook) can quickly cover a considerable proportion of the entire social network within as small as 3 hops and is clearly out of the question for online query processing.

The problem of k-hop reachability cannot be derived from classic reachability, which is actually a special case of k-hop reachability, i.e., when $k = \infty$. Indeed, the k-hop problem is more challenging since more information is required to answer the k-hop reachability query. To see this, consider the *transitive closure* [30] of the adjacency matrix of a directed graph. If we are given this transitive matrix, we can find out whether s can reach t instantly by checking whether their corresponding entry in the transitive matrix is 1 or 0. However, for large graphs, it is infeasible to pre-compute and store the transitive closure, as it takes $O(n^2)$ space, where n is the number of vertices in the graph. Thus, the reachability problem is essentially the problem of how to effectively "encode" the 0–1 transitive matrix into a small index structure which still provides efficient lookup of reachability between any two vertices. For k-hop reachability, the matrix we need to "encode" is no longer a 0–1 matrix. Instead, each entry c_{st} in the matrix contains the length of the shortest path from s to t. Clearly, this matrix contains much more information than the 0–1 transitive matrix. In Sect. 3, we further analyze in detail why all the existing indexes are not suitable for processing k-hop reachability queries.

We propose an efficient index for processing k-hop reachability queries. Our index, called ***k*-reach**, is constructed based on the concept of *vertex cover* [17]. *The main idea of the index design is based on the fact that all vertices in a graph are reachable within 1 hop of some vertex in the vertex cover of the graph.* The vertex cover is small for a wide range of real-world graphs. Thus, we only need to pre-compute k-hop reachability information among a small portion of the vertices, while we also show that it only requires at most 2 bits for encoding each k-hop reachability information.

Another advantage of k-reach is that it allows the inclusion of all high-degree vertices in the vertex cover. This not only gives the same coverage with a smaller vertex cover (and hence reduces the index size), but also allows queries that involve high-degree vertices to be answered more efficiently (e.g., the "Lady Gaga" example given earlier).

The k-reach index may have large storage space if the size of the vertex cover is large. Thus, we also propose a more scalable method to address this problem. The more scalable k-reach index employs a partial coverage as well as relaxes the 1-hop edge coverage in the classic vertex cover to k-hop vertex coverage, thus significantly reducing the index size for handling large graphs.

The k-reach index is simple in design and easy to implement. The index can handle both classic reachability queries and k-hop reachability queries and can even be easily extended to processing shortest path distance queries. We conducted experiments on a wide range of real datasets. Our results show that k-reach is significantly more efficient and scalable than the state-of-the-art shortest path distance indexes [3,25] can be used to answer k-hop reachability queries, thus demonstrating the need for a k-hop reachability index. The results also show that k-reach is also efficient for processing classic reachability queries, even comparable with the state-of-the-art indexes [26,27,33,37] that are primarily designed for classic reachability.

Organization. The rest of the paper is organized as follows. Section 2 formally defines the notations and the problem. Section 3 analyzes the difficulties of applying the existing works for handling k-hop reachability and highlights the challenges. Section 4 presents the details of the k-reach index. Section 5 presents a more scalable index. Section 6 reports the experimental results. Section 7 discusses other related works, followed by the conclusions in Sect. 8.

2 Notations and problem definition

Table 1 lists the notations that are frequently used in this paper.

Let $G = (V, E)$ be a directed graph, where V is the set of vertices and E is the set of edges in G, respectively. An edge $(u, v) \in E$ is a directed edge from u to v, while $(v, u) \in E$ means that the edge is directed from v to u.

Given a pair of vertices s and t in G, we say that t is **reachable** from s, denoted by $s \rightarrow t$, if there exists a simple directed path $P = \langle s, \ldots, t \rangle$ in G. If $|P| \leq k$, where $|P|$ is the path length (i.e., the number of edges on P), then t is also ***k*-hop reachable** from s, denoted by $s \rightarrow_k t$. A **reachability query** is to determine whether $s \rightarrow t$, while a

Table 1 Frequently used notations

Notation	Description
$G = (V, E)$	A directed graph
n or m	The number of vertices or edges in G
$\|G\|$	The size of G, $\|G\| = (n + m)$
$s \to t$	t is reachable from s
$s \to_k t$	t is reachable from s within k hops
$inNei(v, G)$	The set of in-neighbors of v in G
$inDeg(v, G)$	The in-degree of v in G, i.e., $\|inNei(v, G)\|$
$outNei(v, G)$	The set of out-neighbors of v in G
$outDeg(v, G)$	The out-degree of v in G, i.e., $\|outNei(v, G)\|$
$Nei(v, G)$	The set of neighbors of v in G
$Deg(v, G)$	The out-degree of v in G, i.e., $\|Nei(v, G)\|$

***k*-hop reachability query** is to determine whether $s \to_k t$. Note that a reachability query is in fact an ∞-hop reachability query.

Problem definition. *Given a directed graph G, the k-hop reachability indexing problem is to construct an index structure for G to answer k-hop reachability queries.*

The following notations will also be used throughout the paper in the discussion of our indexing and query processing algorithms.

Given a directed graph $G = (V, E)$, define $n = |V|$, $m = |E|$, and $|G| = (n + m)$. We denote the set of **in-neighbors** of a vertex v in G by $inNei(v, G) = \{u : (u, v) \in E\}$, and the **in-degree** of v in G by $inDeg(v, G) = |inNei(v, G)|$. Similarly, we denote the set of **out-neighbors** of v in G by $outNei(v, G) = \{u : (v, u) \in E\}$, and the **out-degree** of v in G by $outDeg(v, G) = |outNei(v, G)|$. We also denote the set of **neighbors** of v in G by $Nei(v, G) = (inNei(v, G) \cup outNei(v, G))$, and the **degree** of v in G by $Deg(v, G) = |Nei(v, G)|$.

3 Reachability versus *k*-hop reachability

In this section, we analyze the suitability of the existing graph reachability indexes [2,5–8,13–16,22,26–28,30,32–34,37,38] for processing k-hop reachability queries. An understanding of these existing works with their relation to the problem of k-hop reachability query processing helps not only the work in this paper but also potential future work along this direction.

We categorize the existing works of reachability indexing into six approaches and then show that they cannot be applied or are inefficient for processing k-hop reachability queries. Note that some existing works may fall into more than one category since they may combine different approaches to solve the problem.

3.1 Directed acyclic-graph-based approach

The first category of indexes is related to *directed acyclic graph* (**DAG**). Many existing indexes for processing reachability queries assume that the input graph is a DAG [6–8,22,26–28,30,33,38], because if the input graph is not a DAG, one can preprocess it and turn it into a DAG as follows. First, compute all the *strongly connected components* (**SCCs**) in the input graph, where an SCC is a maximal subgraph in which there is a path from each vertex to every other vertex in the subgraph. Then, condense each SCC into a single super-vertex, where each super-vertex is a vertex in the DAG. Finally, a directed edge (c_1, c_2) is added from a super-vertex c_1 to another super-vertex c_2 in the DAG iff there exists a directed edge (u, v) in the original graph such that u is in c_1 and v is in c_2 (note that c_1 and c_2 are two SCCs in the original graph).

Condensing a general graph into a DAG can save space, and it works for processing reachability queries since all vertices within an SCC are pairwise reachable from each other. However, for processing k-hop reachability queries, the DAG-based approach fails because two reachable vertices in the DAG may not be k-hop reachable in the original graph, since the shortest path connecting them may have been condensed into a much shorter path (of length $\leq k$) in the DAG. To answer a k-hop reachability query, one has to expand the vertices involved in the DAG to their corresponding SCCs in the original graph in order to examine the k-hop information, which is no cheaper than directly checking k-hop in the original graph.

3.2 Traversal-based vertex coding approach

The second category of graph reachability indexes focuses on designing some *vertex coding scheme based on graph traversal* [2,6,8,32,34,37,38]. A traversal, e.g., *depth-first search* (**DFS**), of a graph assigns each vertex a pair of codes according to the traversal order (e.g., the discovery time and finish time of a vertex in a DFS). The pair of codes obtained from a traversal forms an interval, which can be further modified to capture more information of descendants or of other relevant links. Then, reachability queries can be answered based on the containment relationship of the intervals. Different graph traversal methods may be applied, and there can be multiple traversals depending on the design of the index.

For processing k-hop reachability queries, however, the interval containment test of a traversal-based approach fails to capture the k-hop requirement. To examine the number of hops from the source vertex to the target vertex, one needs to explore the input graph. Although the vertex coding may help guide the exploration, the process can be as expensive as a trivial BFS to process the k-hop reachability query starting from the source vertex.

3.3 Chain-cover-based approach

The third category of graph reachability indexes is constructed based on a *chain cover* of the input graph or partially relied on some chain cover [7,8,22,26–28]. A chain cover of a graph $G = (V, E)$ consists of a set of chains, $\{C_1, \ldots, C_t\}$, where $C_i \subseteq V$, $\bigcup_{1 \le i \le t} C_i = V$ and $(C_i \cap C_j) = \emptyset$, for $1 \le i, j \le t$ and $i \ne j$. For each chain, $C_i = \{v_1, \ldots, v_{c_i}\}$, we have $v_x \rightarrow v_y$ for $1 \le x < y \le c_i$. After computing a chain cover of G, each vertex $v \in V$ is assigned a list of chain codes $\{\sigma_1, \ldots, \sigma_t\}$, where σ_i indicates that v can reach the vertex at the σ_i-th position in the chain C_i. Thus, a reachability query can be answered by examining the lists of chain codes of the vertices involved.

Since a chain or the list of chain codes of a vertex retain only the reachability information between the vertices, the chain-cover-based indexes cannot process a k-hop reachability query. It is not clear how we may extend the chain cover to contain the information of k-hop reachability, since the connections among both the chains and vertices in a chain are all involved. Even though the information of k-hop reachability can be indexed in the chain cover, resolving the inter-connection between chains and intra-connection within a chain to process k-hop reachability can be complicated and expensive.

3.4 2-hop-cover-based approach

The fourth category of works construct reachability indexes based on the concept of *2-hop cover* [5,13–16,27]. The 2-hop cover approach computes for each vertex v in an input graph $G = (V, E)$ two vertex subsets, $L_{in}(v)$ and $L_{out}(v)$, where $L_{in}(v)$ consists of a set of vertices in G that can reach v, and $L_{out}(v)$ consists of a set of vertices in G that can be reached from v. Then, a reachability query is answered as follows: a source vertex s can reach a target vertex t if and only if $(L_{out}(s) \cap L_{in}(t)) \ne \emptyset$.

The 2-hop cover clearly also cannot be used to process k-hop reachability queries because all distance information between the vertices is lost. The 2-hop cover can be extended to encode the distance information of each reachable vertex in $L_{in}(v)$ or $L_{out}(v)$ related to v. However, as shown in many existing works of graph reachability [7,8,26,27], the 2-hop cover has not only a higher complexity but is also significantly less efficient than the recent indexes in real performance for processing reachability queries, not to mention for processing k-hop reachability queries. On the contrary, we show that our approach is efficient for processing both reachability queries and k-hop reachability queries.

3.5 Shortest path approaches

Indexes for processing shortest path or distance queries [3,12,16,25] can be trivially used to process k-hop reachability queries. Shortest path or distance query processing, however, has a significantly higher cost than k-hop reachability query processing, as we will show in our experiments. The 2-hop-cover-based indexes [12,16] are not efficient enough for processing k-hop reachability queries, as explained in Sect. 3.4. The highway-centric labeling approach has an expensive index construction cost and is not scalable and is also shown to be significantly slower than our method in query processing in Sect. 6.4.

Apart from the above-mentioned three indexes, there are also indexes for processing shortest path distance queries in undirected graphs [35,36]. These indexes also have a high indexing cost, and we are not aware how they can be extended to process reachability queries that concern directed graphs.

There are also many indexes developed for processing shortest path and distance queries in planar graphs or road networks (see [1] and the references therein). However, these indexes are specifically optimized for road networks and cannot be applied to directed general graphs.

3.6 Other approaches

Other approaches such as transitive closure [30] can also be extended to encode the k-hop reachability information. However, the transitive closure is in general too large to be practical. A recent work has been proposed to compress the transitive closure using bit vector compression techniques [33], which has shown to be effective for processing reachability queries. However, unlike encoding for graph reachability which requires only boolean indicators, encoding the k-hop reachability information not only requires more bits for each entry, but also breaks the continuity of long sequences of "0"s and "1"s which is crucial for the effectiveness of the bit vector compression techniques [33]. More critically, both transitive closure [30] and compression on transitive closure [33] work only on the much smaller DAG of the input graph, while the DAG-based approach is not applicable for processing k-hop reachability queries as discussed in Sect. 3.1.

4 A vertex-cover-based index

Having discussed the limitations of the existing indexes for processing k-hop reachability queries, in this section, we propose an efficient index, called ***k*-reach**, as a solution.

4.1 K-reach: index construction

The k-reach index is constructed based on the concept of *vertex cover* [17]. We first discuss how to compute a minimum vertex cover of a graph G. Then, we define the index structure and describe the algorithm that constructs the index.

4.1.1 Minimum vertex cover approximation

A set of vertices, S, is a vertex cover of a graph $G = (V, E)$ if for every edge $(u, v) \in E$, we have $(\{u, v\} \cap S) \neq \emptyset$. Obviously, V itself is a vertex cover of G but is too large to be used to construct an index. Thus, we want to minimize the size of the vertex cover.

A vertex cover S is called a *minimum vertex cover* of G if S has the smallest size among all vertex covers of G. The problem of computing the minimum vertex cover is well known to be NP-hard [17]. However, there is a polynomial time algorithm for computing a *2-approximate minimum vertex cover*, which is given as follows.

We randomly select an edge (u, v) from E, add both u and v to S, and then remove u and v from G, together with all edges incident to the two vertices. Note that all edges incident to u or v, whether in-edges or out-edges, can be removed from G because all these edges are covered by either u or v in S. This process is repeated until all edges are removed from G.

The above algorithm takes $O(m + n)$ time since every edge is only touched once. Let C be a minimum vertex cover of G. Then, for every pair of vertices, u and v, selected to be included in S in the above process, either u or v must be in C, because otherwise the edge (u, v) is not covered by any vertex in C. Thus, we have $|S| \leq 2|C|$. From the analysis, we also see that we may simply ignore the direction of the edges in computing a 2-approximate minimum vertex cover of G.

4.1.2 Definition of k-reach and its construction

We now define the structure of the k-reach index as follows.

Definition 1 (*k-reach*) Given a directed graph $G = (V, E)$, a vertex cover S of G, and an integer k, the **k-reach** index of G is a weighted, directed graph $I = (V_I, E_I, \omega_I)$ defined as follows.

- $V_I = S$.
- $E_I = \{(u, v) : u, v \in S, u \rightarrow_k v\}$.
- ω_I is a weight function that assigns a weight to each edge $e = (u, v) \in E_I$ as follows:
  - if $u \rightarrow_{k-2} v$, then $\omega_I(e) = (k - 2)$;
  - else if $u \rightarrow_{k-1} v$, then $\omega_I(e) = (k - 1)$;
  - else (i.e., $u \rightarrow_k v$), then $\omega_I(e) = k$.

Note that in Definition 1, "$u \rightarrow_{k-2} v$" implies "$u \rightarrow_{k-1} v$," both of which also imply "$u \rightarrow_k v$."

Next, we describe the index construction process, as shown in Algorithm 1.

Algorithm 1 first computes a 2-approximate minimum vertex cover, S, of the input graph G by the algorithm given in Sect. 4.1.1. Then, it constructs the graph $I = (V_I, E_I, \omega_I)$ by

Algorithm 1: Construction of k-reach

```
   Input  : A directed graph, $G = (V, E)$; and an integer, $k$
   Output: A $k$-reach index of $G$
1  compute a 2-approximate minimum vertex cover, $S$, of $G$;
2  initialize a weighted, directed graph $I = (V_I, E_I, \omega_I)$;
3  $V_I \leftarrow S$;
4  foreach $u \in S$ do
5      compute $S_k(u) = \{v : v \in S, u \rightarrow_k v\}$ by a $k$-hop BFS from
       $u$;
6      foreach $v \in S_k(u)$ do
7          $E_I \leftarrow (E_I \cup \{(u, v)\})$;
8          if $u \rightarrow_{k-2} v$ then
9              $\omega_I((u, v)) \leftarrow (k - 2)$;
10         else if $u \rightarrow_{k-1} v$ then
11             $\omega_I((u, v)) \leftarrow (k - 1)$;
12         else
13             (
           //*
14         [h]i.e., $u \rightarrow_k v$) $\omega_I((u, v)) \leftarrow k$;
15 return $I = (V_I, E_I, \omega_I)$;
```

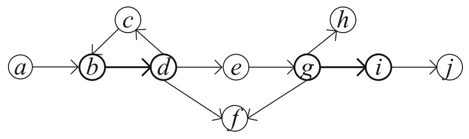
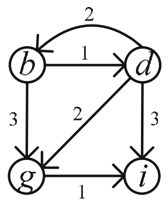


Fig. 1 An example graph G (the vertex cover is $\{b, d, g, i\}$)

Fig. 2 The k-reach graph ($k = 3$), $I = (V_I, E_I, \omega_I)$, of G in Fig. 1



performing a breadth-first search (BFS) of G within k hops from each starting vertex $u \in S$. This process computes the set of all vertices in S that can be reached from u in k hops in G, i.e., the set $S_k(u)$ in Line 5. The rest of the algorithm is simply including each edge (u, v) in E_I, for each $v \in S_k(u)$, and assigning the weight to (u, v).

We give an example of the k-reach index constructed by Algorithm 1 as follows.

Example 1 Given the graph G in Fig. 1. Assume that the 2-approximate algorithm randomly picks the edges, (b, d) and (g, i), in G. Then, $\{b, d, g, i\}$ forms the set of 2-approximate minimum vertex cover of G. We can verify that $\{b, d, g, i\}$ is indeed a vertex cover of G, since every edge in G is incident to at least one of the vertices in $\{b, d, g, i\}$.

Let $k = 3$. The k-reach graph, $I = (V_I, E_I, \omega_I)$, of G is shown in Fig. 2. Since $k = 3$, the possible edge weights are $k - 2 = 1$, $k - 1 = 2$, and $k = 3$. For example, $b \rightarrow_3 g$ in G, and thus, we have the directed edge (b, g) with $\omega_I((b, g)) = 3$ as shown in Fig. 2. We will further explain how we use the k-reach graph to process a k-hop reachability query in Example 2 in Sect. 4.2.

4.1.3 Complexity of constructing k-reach

Computing the 2-approximate minimum vertex cover, S requires $O(m+n)$ time. Constructing the weighted, directed graph $I = (V_I, E_I, \omega_I)$ takes $O(\sum_{u \in S} |G_k(u)|)$ time, where $G_k(u)$ is the subgraph of G that can be reached from u in k hops. Note that it is straightforward to parallelize this process if more machines or CPU cores are available.

The size of the index, i.e., the size of the graph I, depends on both the size of S and $S_k(u)$ for each $u \in S$. However, it is difficult to derive a theoretical bound of S or $S_k(u)$ for real-world graphs since they often vary significantly in characteristics with respect to S and $S_k(u)$, even though some graphs may share some similar properties such as sparsity and power-law degree distribution. We are not aware of any existing work that gives a theoretical bound on the size of the minimum vertex cover for real-world graphs. Thus, we examine the size of the index experimentally for a wide range of real-world graphs.

Finally, constructing the k-reach index uses $O(m + n)$ memory space. Note that the constructed index is then stored on disk simply in the adjacency list representation [17] of the k-reach graph I.

4.2 K-reach: query processing

We now discuss how we process a k-hop reachability query using the k-reach index.

4.2.1 Query processing using k-reach

We give the algorithm for query processing using k-reach in Algorithm 2.

Given two query vertices, s and t, Algorithm 2 processes the k-hop reachability query by considering four cases. The following theorem proves the correctness of the algorithm for processing a k-hop reachability query using the k-reach index. The proof also explains how a query is processed.

Theorem 1 *Given a directed graph, $G = (V, E)$, the k-reach index, $I = (V_I, E_I, \omega_I)$, of G, and two query vertices, s and t, Algorithm 2 returns* `true` *if $s \to_k t$ in G and* `false` *otherwise.*

Proof Note that V_I is a vertex cover of G. There are only four possible cases in processing a k-hop reachability query by considering the membership of s and t in V_I. Algorithm 2 processes the query according to which case the query belongs to as follows.

Algorithm 2: Query processing using k-reach

```
Input  : A directed graph, $G = (V, E)$; a $k$-reach index,
         $I = (V_I, E_I, \omega_I)$, of $G$; and two query vertices, $s$ and $t$
Output: A boolean indicator whether $s \to_k t$
   //case 1: both $s$ and $t$ are in the vertex cover
1  if $s \in V_I$ and $t \in V_I$ then
2      if $(s, t) \in E_I$ then
3          return true;
4      else
5          return false;
   //case 2: only $s$ is in the vertex cover
6  else if $s \in V_I$ and $t \notin V_I$ then
7      if $\exists v \in inNei(t, G)$ such that $(s, v) \in E_I$ and
       $\omega_I((s, v)) \leq (k-1)$ then
8          return true;
9      else
10         return false;
   //case 3: only $t$ is in the vertex cover
11 else if $s \notin V_I$ and $t \in V_I$ then
12     if $\exists v \in outNei(s, G)$ such that $(v, t) \in E_I$ and
       $\omega_I((v, t)) \leq (k-1)$ then
13         return true;
14     else
15         return false;
   //case 4: both $s$ and $t$ are not in the vertex cover
16 else if $s \notin V_I$ and $t \notin V_I$ then
17     if $\exists u \in outNei(s, G)$ and $\exists v \in inNei(t, G)$ such that
       $(u, v) \in E_I$ and $\omega_I((u, v)) \leq (k-2)$ then
18         return true;
19     else
20         return false;
```

Case 1: both s and t are in V_I. In this case, if $s \to_k t$ in G, then the edge (s, t) must exist in I. Thus, the answer to the query by Algorithm 2 is trivially correct.

Case 2: only s is in V_I. In this case, all in-neighbors (if any) of t must be in V_I. Otherwise (i.e., if $\exists v \in inNei(t, G)$ such that v is not in V_I), then the edge (v, t) is not covered since both v and t are not in the vertex cover V_I. Thus, if $s \to_k t$ in G, then there must exist an in-neighbor v of t such that $\omega_I((s, v)) \leq (k-1)$, since the (directed) path from s to t must pass through at least one in-neighbor of t. Therefore, it is sufficient to check whether $(s, v) \in E_I$ and $\omega_I((s, v)) \leq (k-1)$ in order to determine whether $s \to_k t$.

Case 3: only t is in V_I. This case is similar to Case 2. Now since s is not in the vertex cover V_I, all out-neighbors (if any) of s must be in V_I; otherwise, the edge (s, v) is not covered for some $v \in outNei(s, G)$ and $v \notin V_I$. Thus, similar to Case 2, it is sufficient to check whether $(v, t) \in E_I$ and $\omega_I((v, t)) \leq (k-1)$ in order to determine whether $s \to_k t$.

Case 4: both s and t are not in V_I. In this case, all out-neighbors (if any) of s and all in-neighbors (if any) of t must be in V_I; otherwise, the edges (s, u) and (v, t) are not covered for some $u \in outNei(s, G)$, $v \in inNei(t, G)$, and $u, v \notin V_I$. Thus, if $s \rightarrow_k t$ in G, then there must exist an out-neighbor u of s and an in-neighbor v of t such that $\omega_I((u, v)) \leq (k - 2)$, since the (directed) path from s to t must first go from s to some $u \in outNei(s, G)$, and finally pass through some $v \in inNei(t, G)$ to t. Therefore, it is sufficient to check whether $(u, v) \in E_I$ and $\omega_I((u, v)) \leq (k-2)$ in order to determine whether $s \rightarrow_k t$. □

We give an example of using the k-reach index to process k-hop reachability queries as follows. We use $s \nrightarrow_k t$ to indicate that t is **not k-hop reachable** from s.

Example 2 Consider the graph G in Fig. 1 and the k-reach graph $I = (V_I, E_I, \omega_I)$ of G in Fig. 2, where $k = 3$. We discuss how we use k-reach to process each of four cases in Algorithm 2 as follows.

Case 1: both s and t are in V_I. Let $s = b \in V_I$. We first consider $t = g \in V_I$. Since $(b, g) \in E_I$, we have $b \rightarrow_k g$. However, if $t = i \in V_I$, then $b \nrightarrow_k i$ since $(b, i) \notin E_I$, although b can reach i in G (but in $4 > k = 3$ hops).

Case 2: only s is in V_I. Let $s = d \in V_I$. If $t = h \notin V_I$, then we have $d \rightarrow_k h$ since there is an in-neighbor g of h such that $(d, g) \in E_I$ with $\omega_I((d, g)) = 2 \leq (k - 1) = 2$. But if $t = j \notin V_I$, then $d \nrightarrow_k j$ since for the only in-neighbor i of j, although $(d, i) \in E_I$, we have $\omega_I((d, i)) = 3 > (k - 1)$. We can easily verify in G that j is reachable from d in at least 4 hops and thus not 3-hop reachable from d.

Case 3: only t is in V_I. Let $s = a \notin V_I$. If $t = d \in V_I$, we have $a \rightarrow_k d$ since there is an out-neighbor b of a such that $(b, d) \in E_I$ with $\omega_I((b, d)) = 1 \leq (k - 1) = 2$. But if $t = g \in V_I$, then $a \nrightarrow_k g$ since $\omega_I((b, g)) = 3 > (k - 1)$. We can easily verify in G that g is reachable from a in at least 4 hops and thus not 3-hop reachable from a.

Case 4: both s and t are not in V_I. Let $s = c \notin V_I$. If $t = f \notin V_I$, we have $c \rightarrow_k f$ since there is an out-neighbor b of c and an in-neighbor d of f such that $(b, d) \in E_I$ with $\omega_I((b, d)) = 1 \leq (k - 2) = 1$. But if $t = h \notin V_I$, then $c \nrightarrow_k h$ since h has only one in-neighbor g but $\omega_I((b, g)) = 3 > (k - 2)$. We can easily verify in G that h is reachable from c in at least 5 hops and thus not 3-hop reachable from c. □

4.2.2 *Complexity of query processing using k-reach*

Since existing indexes for reachability querying are all in-memory indexes, for fairness of comparison in query processing, we first load the k-reach graph I into main memory and keep the graph in its adjacency list representation. We also keep the original graph in memory and use a bit vector to indicate whether each $v \in V$ belongs to V_I. Then, the membership tests whether s and t belong to V_I take $O(1)$ time by directly accessing the bit vector in memory. Checking whether an edge (u, v) exists in E_I and retrieving its weight take $O(\log outDeg(u, I))$ or $O(\log inDeg(v, I))$ CPU time, by a binary search of v in the adjacency list of u or a binary search of u in the adjacency list of v.

Thus, Case 1 of Algorithm 2 takes $O(\log outDeg(s, I))$ time, Case 2 takes $O(outDeg(s, I) + inDeg(t, G))$ time, Case 3 takes $O(outDeg(s, G) + inDeg(t, I))$ time, Case 4 uses $O(\sum_{u \in outNei(s,G)}(outDeg(u, I) + inDeg(t, G)))$ time. Note that for Cases 2 to 4 we can perform intersection of the involved adjacency lists and terminate earlier as soon as an edge is found to give a `true` answer.

4.3 The curse of high-degree vertices

According to the complexity analysis of query processing in Sect. 4.2.2, the query performance depends largely on the degree of a vertex in G and in I. Many large real-world graphs have a power-law degree distribution, and hence, a small number of vertices may have a very high degree. For example, the singer-songwriter Lady Gaga has 40,000,000 fans on Facebook. Therefore, it is crucial to avoid having these high-degree vertices as query vertices that fall into Case 4, or even Cases 2 and 3, of Algorithm 2. Nevertheless, statistically, these high-degree vertices may indeed have a higher probability to be picked as query vertices since they usually represent objects that attract more attention.

To enable these high-degree query vertices to be processed efficiently, we modify the algorithm for computing the 2-approximate minimum vertex cover in Sect. 4.1.1 as follows. In picking an edge in order to include its two end vertices in the vertex cover, we give a higher priority to edges with either or both end vertices that have a high degree. Since most real-world graphs have only a very small percentage of high-degree vertices [29], we can easily include all such vertices in the vertex cover without sacrificing the approximation ratio. In fact, including the high-degree vertices in the vertex cover is a greedy strategy that tends to reduce the size of the vertex cover in practice, since a high-degree vertex covers more edges than a low-degree one.

Prior study has shown that for a typical real-world graph with power-law degree distribution, if the graph has 1 million vertices, then the "h-index" of the graph is only about 300 [10]; that is, the 1 million-vertex graph contains only about $h = 300$ vertices with degree at least $h = 300$.

The vertices that have a high degree in G, however, also tend to have a high degree in I. This not only reflects a trade off in query performance but also increases the index size. However, this problem can be alleviated as follows. Since there are only three types of edge weight, i.e., k, $(k-1)$, and $(k-2)$, in I, we only need to use 2 bits to represent each edge weight. Thus, the set of neighbors of those high-degree vertices in I can be effectively represented in a more compact way, such as *interval lists* or *partitioned word aligned* *hybrid compression* [33], which have been shown effective for reducing the storage size of the edge-transitive closure graph for processing reachability queries. Note that with the compact representations, we only need to locate the corresponding interval or bits for query processing [33], instead of searching the list of neighbors.

4.4 A general k

We next consider whether one wants to ask k-hop reachability queries for different values of k. In this case, a specific k-reach index (i.e., the index is built on a specific value of k) is not able to handle a general k. However, we note that if the index can process k-hop reachability queries with a general k, then the index is essentially an index for shortest path distance queries.

Our index can be easily generalized to process shortest path distance queries by keeping the distance information between any two vertices in the vertex cover, which can be computed by doing a full BFS instead of a k-hop BFS in Line 5 of Algorithm 1. This requires $\lg \delta$ bits for each edge weight (instead of 2 bits as with a specific k), where δ is the diameter of the input graph.

5 Answering k-hop reachability queries in large graphs

The vertex-cover-based index proposed in Sect. 4 is efficient for processing both classic reachability queries and k-hop reachability queries when the input graph is not large, as we will verify by experiments. However, when the input graph is large, the size of the vertex cover is often not small for many real-world graphs. When the vertex cover is large, the vertex-cover-based index will have a high indexing cost and may not scale. Although the index construction process, i.e., the k-hop BFS from vertices in the vertex cover, can be parallelized, the storage requirement may still be high for large graphs.

To address the scalability problem of the vertex-cover-based index, we propose a partial vertex cover to trade query processing time for index storage space.

5.1 Overview

We first give an overview on the design of our index to be presented in the following three subsections:

1. First in Sect. 5.2 we introduce a one-level index constructed based on a partial vertex cover with maximum coverage of vertices and identify its weaknesses.
2. Then in Sect. 5.3 we present a two-level index, which alleviates the problem of the one-level index.
3. Finally, in Sect. 5.4 we improve the two-level index by relaxing the coverage of the partial vertex cover (at both levels) from 1 hop to k hops, thus giving significantly greater coverage and hence a more scalable index.

5.2 Partial vertex cover with maximum coverage

We first introduce a partial vertex cover with maximum coverage of vertices in a graph and then apply it to construct a simple version of a more scalable k-reach index.

Given a fixed number or a budget b, we want to compute a **partial vertex cover**, S, of size b such that S covers the largest number of vertices in G. Formally, the problem is to find a subset $S \subseteq V$, such that $|S| \leq b$ and $|\bigcup_{v \in S} Nei(v, G)|$ is maximized.

The problem is in fact equivalent to the maximum coverage problem, which is NP-hard and cannot be approximated within $(1 - \frac{1}{e} + o(1)) \approx 0.632$ under standard assumptions [21].

We compute a partial vertex cover S with maximum coverage by a greedy process as follows. At each iteration, we select the vertex with the highest degree (ties are broken arbitrarily) into S, then we remove the vertex and all edges incident to it from G, and repeat the process until $|S| = b$.

This greedy algorithm is an approximation algorithm, which has an approximation ratio of $(1 - \frac{1}{e})$ [21]. Moreover, it is also shown in [20] that the greedy algorithm is the best-possible polynomial time approximation algorithm for maximum coverage.

We now define a one-level index that is constructed based on a partial vertex cover.

Definition 2 (*k-reach with partial coverage*) Given a directed graph $G = (V, E)$, a partial vertex cover S of G, and an integer k, the k-reach index of G with respect to S is a weighted, directed graph $D = (V_D, E_D, \omega_D)$ defined as follows.

- $V_D = \{v : v \in V, u \in S, u \rightarrow_k v \text{ or } v \rightarrow_k u\}$.
- $E_D = \{(u, v) : u \in S, v \in V_D, u \rightarrow_k v\} \cup \{(v, u) : u \in S, v \in V_D, v \rightarrow_k u\}$.
- ω_D is a weight function that assigns a weight, $d(u, v)$, to each edge $e = (u, v) \in E_D$, where $d(u, v)$ is the shortest path distance from u to v in G.

In Definition 2, we assume that $v \rightarrow_0 v$, i.e., v reaches v itself in 0 hop, and hence $S \subseteq V_D$. But note that the edge (v, v) needs not to be added to E_D as to save space.

Intuitively, the graph D keeps the set of vertices that are within k hops from or to some vertex in S, as well as their shortest path distance information (encoded by edge weight).

The idea of using D to answer k-hop reachability queries is as follows. Given two query vertices, s and t, we have two cases: (1) if $t \in outNei(s, D)$ or $s \in inNei(t, D)$, then $s \rightarrow_k t$; or (2) if there exist $v \in (outNei(s, D) \cap inNei(t, D))$ such that $(d(s, v) + d(v, t)) \leq k$, then $s \rightarrow_k t$.

However, since the coverage by S may be only partial, it is possible that neither (1) nor (2) is true, but we still have $s \rightarrow_k t$. Thus, we will need to traverse the graph G by BFS to determine whether $s \rightarrow_k t$ if both cases (1) and (2) fail.

In the worst case, the space complexity is $O(|S||V|)$. Thus, we can only afford a small partial vertex cover S for a large graph. However, there is always a tradeoff here: a smaller partial vertex cover means that more queries do not fall into cases (1) and (2) and are hence answered by traversing the graph G, which leads to a higher cost for query processing; on the other hand, a larger partial vertex cover allows us to answer more queries efficiently by cases (1) and (2), but it implies a larger index size as well as higher index construction cost.

To address the above-mentioned problem, we introduce a two-level index in the following subsection.

5.3 A two-level index

The main idea of the two-level index is: (1) first compute a small partial vertex cover S with maximum coverage and construct the k-reach index of G with respect to S; (2) then remove S (together with all edges incident to vertices in S) from G to obtain a smaller graph G'; and (3) compute a second k-reach index of G' with respect to a partial vertex cover of G'.

The intuition behind the two-level k-reach index is that many real-world graphs consist of only a small number of high-degree vertices, and the removal of these vertices will significantly reduce the size of the graph, so that the index construction cost is lowered at the first level with a smaller partial vertex cover while more coverage is attained at the second level in a smaller graph G'.

We formally define the two-level k-reach index as follows.

Definition 3 (*Two-level k-reach*) Given a directed graph $G = (V, E)$ and an integer k, a **two-level k-reach index** of G consists of two weighted, directed graphs, $D_1 = (V_{D_1}, E_{D_1}, \omega_{D_1})$ and $D_2 = (V_{D_2}, E_{D_2}, \omega_{D_2})$, defined as follows.

Given a partial vertex cover S of G, $D_1 = (V_{D_1}, E_{D_1}, \omega_{D_1})$ is the k-reach index of G with respect to S.

Let $G' = (V', E')$ be the graph obtained after removing S and all edges incident to S from G. Let S' be a partial vertex cover of G', $D_2 = (V_{D_2}, E_{D_2}, \omega_{D_2})$ is the k-reach index of G' with respect to S'.

With another level added to the index, query evaluation also needs to be processed at both levels. If space is limited such that we cannot build a k-reach index with full coverage, and given the same available space, the two-level k-reach index in general gives better query performance than the one-level k-reach index due to better coverage (i.e., less queries are answered by graph traversal).

However, the same problem mentioned in Sect. 5.2 that exists in the one-level k-reach index still exists in the graph D_2 at the second level, though to a lesser degree; that is, we still require $O(|S'||V_{G'}|)$ space at the second level. Thus, the index size can still be too large and index construction too costly for the method to scale to handle large graphs. To make the index more scalable, we propose a relaxed vertex cover as well as some optimization techniques in the following subsection.

5.4 A more scalable k-reach index

We present techniques that we apply to reduce the size of our index in order to process large graphs.

5.4.1 k-relaxed partial vertex cover

Given a fixed number or a budget b and an integer k, a ***k*-relaxed partial vertex cover** with maximum coverage is a subset, S, such that $|S| \leq b$ and S covers the largest number of vertices that are within k hops of any vertex in S. Formally, the problem is to find a subset $S \subseteq V$, such that $|S| \leq b$ and $|\bigcup_{v \in S}\{u : v \rightarrow_k u\}|$ is maximized.

Intuitively, a k-relaxed partial vertex cover relaxes the coverage from 1 hop in a partial vertex cover to k hops. Thus, given the same budget b, a k-relaxed partial vertex cover can cover significantly more vertices than a partial vertex cover. Or from another angle, we can attain the same coverage using a much smaller budget. As a result, with the same computing resources (e.g., available main memory and storage space), using a k-relaxed partial vertex cover in place of a partial vertex cover allows us to handle much larger graphs.

Note that this k-relaxed partial vertex cover is different from the h-hop vertex cover proposed in [11], as the former covers vertices while the latter must cover every edge within h hops. Note that for the special case when $k = h = 1$, the k-relaxed (full) vertex cover is in fact the dominating set, while the h-hop vertex cover is the vertex cover. It is well known that the minimum dominating set is significantly smaller than the minimum vertex cover in general. Thus, with the same set of vertices, the k-relaxed partial vertex cover can have significantly greater coverage than the h-hop vertex cover.

5.4.2 Removal of redundant edges

A redundant edge in either D_1 or D_2 (defined in Definition 3) is an edge without which query processing using D_1 or D_2 returns the same answer. We may apply triangle inequality to remove redundant edges as follows.

Let D be a k-reach graph with partial coverage. For each edge $(u, v) \in E_D$, if there exist two edges $(u, w), (w, v) \in E_D$ such that $d(u, w) + d(w, v) \leq d(u, v)$, then the edge (u, v) is redundant and can be removed from D.

Removing redundant edges by triangle inequality, however, is expensive since an intersection is needed between $outNei(u, D)$ and $inNei(v, D)$ for each edge $(u, v) \in E_D$. In our algorithm (to be presented in Sect. 5.5), we incorporate edge removal in the process of constructing D to avoid expensive triangle inequality checking.

5.5 Algorithms and complexity

We now present the algorithms for constructing a more scalable k-reach index and query processing using the index, followed by an analysis of their complexity.

We first give the algorithm for index construction, as shown in Algorithm 3. The algorithm consists of mainly two parts: Lines 1–10 for the construction of D_1 at the first level and Lines 11–22 for the construction of D_2 at the second level. In addition, Lines 23–24 also obtain a much smaller residual graph D_3, so that queries involving uncovered vertices can be processed in a small graph instead of the large input graph.

To construct D_1, the algorithm greedily picks the vertex v that has the largest uncovered neighbors, until b vertices are selected into S. Then, a BFS is started from v in G for k hops. For any vertex u visited within the k hops, we process u as follows: (1) if u has already been selected in S, we simply add the edge (v, u) to D_1, with the edge weight $\omega_{D_1}((v, u)) = d(v, u)$, where $d(v, u)$ is the shortest path distance from v to u computed by the BFS; or else (2) if all vertices on the path of the BFS expansion from v to u (except v) are in S, we add u and the edge (v, u) to D_1 and mark u as covered. The above 2 steps are then applied in a similar way to the reverse graph of G, in order to process reachability from u to v. Then, we add v to S and mark v as covered.

During the k-hop BFS starting from v, when we visit u, the edge (v, u) is redundant in D_1 if there exists some vertex $w \in S$ on the path from v to u, i.e., w is a BFS predecessor of u, since $d(v, u)$ can be obtained by $(d(v, w) + d(w, u))$. Note that $w \in S$, and thus, both $d(v, w)$ and $d(w, u)$ are in D_1, where w is the nearest BFS predecessor of u.

After computing S and the graph D_1, we remove S, together with all edges incident to S, from G to obtain a smaller graph, named as G'. Then, we compute S' and D_2 from G' in the same way as we compute S and D_1 from G, except that the size of S' and hence D_2 are determined by the available main memory size, i.e., we stop the process until main memory is not sufficient to hold the index or until all vertices have been already covered.

Finally, we remove S', together with all edges incident to S', from G' to obtain a residual graph D_3. Thus, D_3 is often a significantly smaller and sparser than G.

Next, we present our algorithm for query processing using the scalable k-reach index constructed by Algorithm 3. As shown in Algorithm 4, a query may fall into one of the five cases. We discuss each case as follows.

Case 1 is simple, for which we only need to check the existence of an edge in D_1 to answer a query. Case 2 processes a query by first checking the existence of the edge (s, t) in D_1; if (s, t) does not exist, we then check if there exists a vertex v as an out-neighbor of s and an in-neighbor of t, such that $\omega_{D_1}((s, v)) + \omega_{D_1}((v, t)) \leq k$. Case 3 first checks the existence of the edge (s, t) in D_2 and if (s, t) does not exist, it further checks whether there exist an out-neighbor u of s and

Algorithm 3: Construction of scalable k-reach

```
Input : A directed graph, $G = (V, E)$; an integer, $k$; and a
        budget, $b$
Output: A more scalable $k$-reach index of $G$, consisting of
        $D_1 = (V_{D_1}, E_{D_1}, \omega_{D_1})$, $D_2 = (V_{D_2}, E_{D_2}, \omega_{D_2})$, and a
        residual graph $D_3$
   //construction of $D_1$ at the first level
1  mark all vertices in $V$ uncovered;
2  $S \leftarrow \emptyset$, $V_{D_1} \leftarrow \emptyset$, $E_{D_1} \leftarrow \emptyset$;
3  let $G_r$ be the reverse graph of $G$, i.e., edge $(v, u)$ in $G_r$ iff edge
   $(u, v)$ in $G$;
4  $i \leftarrow 1$;
5  while $i \leq b$ do
6      select the vertex $v$ that has the largest number of uncovered
       neighbors;
7      start a $k$-hop BFS from $v$ in $G$, for any vertex $u$ visited: if
       $u \in S$, then add $(v, u)$ to $E_{D_1}$ with $\omega_{D_1}((v, u)) = d(v, u)$;
       else if none of $u$'s predecessors (except $v$) are in $S$, then add $u$
       to $V_{D_1}$ and $(v, u)$ to $E_{D_1}$ with $\omega_{D_1}((v, u)) = d(v, u)$, and
       mark $u$ as covered;
8      process Line 3 with "$G$ replaced by $G_r$" and "$(v, u)$ replaced
       by $(u, v)$";
9      add $v$ to $S$ and mark $v$ as covered;
10     $i \leftarrow i + 1$;
   //construction of $D_2$ at the second level
11 remove $S$, and all edges incident to vertices in $S$, from $G$, and
   name the remaining graph as $G'$;
12 let $G'_r$ be the reverse graph of $G'$;
13 $S' \leftarrow \emptyset$, $V_{D_2} \leftarrow \emptyset$, $E_{D_2} \leftarrow \emptyset$;
14 while some vertices in G' are not covered and main memory is
   not used up do
15     process Lines 3-3 with "$G$, $G_r$, $S$, $V_{D_1}$, and $E_{D_1}$" replaced by
       $G'$, $G'_r$, $S'$, $V_{D_2}$, and $E_{D_2}$," respectively;
16 foreach pair $(u, v) \in (V_{D_1} \cap S')$ do
17     if $(u, v) \in E_{D_2}$ then
18         $\omega_{D_2}((u, v)) \leftarrow \min\{\omega_{D_2}((u, v)), \min\{\omega_{D_1}((u, x)) +$
           $\omega_{D_1}((x, y)) + \omega_{D_1}((y, v)) : x, y \in S\}\}$, where
           $\omega_{D_1}((x, y)) = 0$ if $x = y$;
19     else
20         $d \leftarrow \min\{\omega_{D_1}((u, x)) + \omega_{D_1}((x, y)) + \omega_{D_1}((y, v)) :$
           $x, y \in S\}$, where $\omega_{D_1}((x, y)) = 0$ if $x = y$;
21         if $d \leq k$ then
22             add $(u, v)$ to $E_{D_2}$ with $\omega_{D_2}((u, v)) = d$;
   //construction of the residual graph $D_3$
23 remove $S'$, and all edges incident to vertices in $S'$, from $G'$, and
   name the remaining graph as $D_3$;
24 return $D_1 = (V_{D_1}, E_{D_1}, \omega_{D_1})$, $D_2 = (V_{D_2}, E_{D_2}, \omega_{D_2})$, and $D_3$;
```

Algorithm 4: Query processing using scalable k-reach

```
Input : A more scalable $k$-reach index of $G$, consisting of
        $D_1 = (V_{D_1}, E_{D_1}, \omega_{D_1})$, $D_2 = (V_{D_2}, E_{D_2}, \omega_{D_2})$, a
        residual graph $D_3$, and two query vertices, $s$ and $t$
Output: A boolean indicator whether $s \rightarrow_k t$

   //case 1: both $s$ and $t$ are in $S$
1  if $s \in S$ and $t \in S$ then
2      if $(s, t) \in E_{D_1}$ then
3          return true;
4      else
5          return false;

   //case 2: $s$ or $t$ is in $S$, but not both
6  else if $s \in S$ or $t \in S$ then
7      if $(s, t) \in E_{D_1}$, or $\exists v \in S$ such that
       $\omega_{D_1}((s, v)) + \omega_{D_1}((v, t)) \leq k$, then
8          return true;
9      else
10         return false;

   //case 3: both $s$ and $t$ are in $S'$
11 else if $s \in S'$ and $t \in S'$ then
12     if $(s, t) \in E_{D_2}$, or $\exists u, v \in S$ such that
       $\omega_{D_1}((s, u)) + \omega_{D_1}((u, v)) + \omega_{D_1}((v, t)) \leq k$, then
13         return true;
14     else
15         return false;

   //case 4: $s, t \notin S$ and $s$ or $t$ is in $S'$
16 else if $s \in S'$ or $t \in S'$ then
17     if $(s, t) \in E_{D_2}$, or $\exists v \in S'$ such that
       $\omega_{D_2}((s, v)) + \omega_{D_2}((v, t)) \leq k$, or $\exists u, v \in S$ such that
       $\omega_{D_1}((s, u)) + \omega_{D_1}((u, v)) + \omega_{D_1}((v, t)) \leq k$, then
18         return true;
19     else
20         return false;

   //case 5: both $s$ and $t$ are not $S$ or $S'$
21 else if $s, t \in V_{D_3}$ then
22     if $\exists u, v \in S$ such that
       $\omega_{D_1}((s, u)) + \omega_{D_1}((u, v)) + \omega_{D_1}((v, t)) \leq k$,
23     or $\exists u, v \in S'$ such that
       $\omega_{D_2}((s, u)) + \omega_{D_2}((u, v)) + \omega_{D_2}((v, t)) \leq k$ then
24         return true;
25     else
26         start BFS from $s$ in $D_3$ and from $t$ in the reverse graph of
           $D_3$ in parallel for $\lceil k/2 \rceil$ hops: if the two BFSs meet at a
           common vertex $v$ and $d(s, v) + d(t, v) \leq k$, return
           true; otherwise, return false;
```

an in-neighbor v of t such that $\omega_{D_1}((s, u)) + \omega_{D_1}((u, v)) + \omega_{D_1}((v, t)) \leq k$, where $\omega_{D_1}((u, v)) = 0$ if $u = v$. Case 4 is Case 3 but it may also check whether there exists a vertex v as an out-neighbor of s and an in-neighbor of t in D_2, such that $\omega_{D_2}((s, v)) + \omega_{D_2}((v, t)) \leq k$. Lastly, Case 5 first checks whether there exists an out-neighbor u of s and an in-neighbor v of t in either D_1 or D_2, and if the sum of the weights of the edges (s, u), (u, v) and (v, t) is greater than k, then the query is answered by BFS from s and t in parallel to see if they can meet at a common vertex in $\lceil k/2 \rceil$ hops.

Table 2 The five cases of query processing in Algorithm 4

	S	S'	V_{D_3}
Case 1	s, t		
Case 2	s	t	
Case 2	s		t
Case 2	t	s	
Case 3		s, t	
Case 4		s	t
Case 2	t		s
Case 4		t	s
Case 5			s, t

The following theorem proves the correctness of the algorithm for processing a k-hop reachability query by Algorithm 4. The proof also presents details about how a query is processed by Algorithm 4.

Theorem 2 *Given D_1, D_2 and D_3, and two query vertices, s and t, Algorithm 4 returns* `true` *if $s \rightarrow_k t$ in G and* `false` *otherwise.*

Proof According to the index construction shown in Algorithm 3, the vertex set V can be divided into three disjoint subsets, S, S', and V_{D_3}. A query vertex, s or t, may belong to one of these three subsets, and hence, there are 9 possible combinations as shown in Table 2. The table also maps each of the 9 possible combinations to one of the 5 cases of query processing presented in Algorithm 4. We then prove the correctness for each case as follows.

Case 1: If $s, t \in S$, then both (s, t) and (t, s) are in E_{D_1} if and only if $s \rightarrow_k t$ in G.

Case 2: Without loss of generality, let us assume $s \in S$ and $t \notin S$, i.e., $t \in S'$ or $t \in V_{D_3}$. If $(s, t) \in E_{D_1}$, then by the construction of D_1, we have $s \rightarrow_k t$ in G. Otherwise, if $s \rightarrow_k t$ in G, then there must exist some $v \in S$ such that v is a predecessor of t when a k-hop BFS is started from s, and thus the edge (s, t) is considered redundant and not created in D_1 (see Line 3 of Algorithm 3). Since $\forall x, y \in S$, $(x, y) \in E_{D_1}$ if $x \rightarrow_k y$ in G, and v is a predecessor of t in the BFS from s to t, we have $s \rightarrow_k t$ in G if $\exists v \in S$ such that $\omega_{D_1}((s, v)) + \omega_{D_1}((v, t)) \leq k$.

Case 3: If $s \rightarrow_k t$ in G' (and hence also in G), then both (s, t) and (t, s) are in E_{D_2} since $s, t \in S'$. Else if $s \rightarrow_k t$ in G but not in G', then s must reach t through some $u, v \in S$, which are not in G', where $\omega_{D_1}((u, v)) = 0$ if $u = v$. Note that if $u \neq v$, then (s, u), (u, v) and (v, t) must be in E_{D_1}, since $s \rightarrow_k t$ in G.

Case 4: If $s \rightarrow_k t$ in G', we have either $(s, t) \in E_{D_2}$ or s reaches t via some $v \in S'$ since only one of s and t is in S'

and the other is in V_{D_3}. Else if $s \rightarrow_k t$ in G but not in G', then s must reach t through some $u, v \in S$ as discussed in Case 3.

Case 5: If $s \rightarrow_k t$ in G, we have the following 3 cases: (1) s reaches t through some $u, v \in S$ within k hops in G; or (2) s reaches t through some $u, v \in S'$ within k hops in G' (and hence also in G); or (3) s reaches t in D_3 within k hops (and hence also in G). All the three cases are handled by Algorithm 4. Note that if s reaches t via u and v, then both u and v must be either in S or in S'. The reason is: suppose $u \in S$, then we have either $u = v$ or there must exist some $v \in S$ since $s \rightarrow_k t$ in G (and similarly for $u, v \in S'$). □

We now analyze the complexity of our algorithms.

The complexity (in terms of both time and storage space) of constructing the more scalable k-reach index (i.e., Algorithm 3) is clearly bounded by that of constructing a basic k-reach index (i.e., Algorithm 1), since the more scalable version is a partial and relaxed version of the basic k-reach index, with the second level (i.e., D_2) even constructed from a smaller subgraph G' of G.

As discussed for the k-reach graph I in Sect. 4.2.2, for query processing, we also first load D_1, D_2, and D_3 into main memory and keep the graphs in their adjacency list representation. The complexity of query processing by Algorithm 4 then depends on which case a query is processed.

For Case 1, it takes $O(\log outDeg(s, D_1))$ or $O(\log inDeg(t, D_1))$ time, where D_1 is stored in the adjacency list representation. For Case 2, it intersects $outNei(s, D_1)$ and $inNei(t, D_1)$ to find the first common vertex v, which takes $O(outDeg(s, D_1) + inDeg(t, D_1))$ time in the worst case. For Case 3, it requires to join $outNei(s, D_1)$ and $inNei(t, D_1)$, which takes $O(outDeg(s, D_1) \times inDeg(t, D_1))$ time. For Case 4, it adds $O(outDeg(s, D_2) + inDeg(t, D_2))$ time to the time of Case 3. For Case 5, it has two joins in Line 5 of Algorithm 4, which takes $O(outDeg(s, D_1) \times inDeg(t, D_1) + outDeg(s, D_2) \times inDeg(t, D_2))$, while the k-hop BFS in Line 5 takes $O(|V_{D_3}| + |E_{D_3}|)$ time in the worst case.

The size of the graph D_1 depends on the value of the budget b, which can be specified by a user according to the available memory or storage space. We set the default value of b as the "h-index" of a graph, i.e., the maximum h such that h vertices in the graph have degree at least h. For a power-law graph, the value of h can be approximated by $h \leq n^{\frac{\mathcal{R}}{\mathcal{R}-1}}$ [10], where $\mathcal{R}$ is a constant between −0.8 and −0.7 for a typical power-law graph [19].

The size of the graph D_2 depends on the available main memory. The more memory that can be allocated to D_2 is, the greater is the coverage and also the smaller the residual graph D_3. In addition, the size of D_3 can be approximated by $(m - \sum_{i=1}^{j} (\frac{i}{n})^{\mathcal{R}})$, where $j = |S| + |S'|$.

5.6 An adaptive k-reach index

We combine the basic k-reach index proposed in Sect. 4 and the more scalable version proposed in this section as follows. Given a graph G, we first compute an approximate minimum vertex cover, S, of G. Given S, we can estimate the upper bound of the size of the basic k-reach index as $|S|^2$. If the size $|S|^2$ is affordable with the available computing resource, we construct the basic k-reach index by Algorithm 1; otherwise, we construct the more scalable k-reach index by Algorithm 3. Then, queries are processed by Algorithms 2 or 4 depending on which version of k-reach is constructed. According to Sect. 4.1.1, constructing the approximate minimum vertex cover S only requires a scan of the graph once, which is only an insignificant portion of the overall index construction cost.

5.7 Extending to general k

In Sect. 4.4, we have discussed how the basic k-reach index proposed in Sect. 4 may be extended to handle different values of k. We now discuss how the more scalable version proposed in this section may be extended to handle any value of k.

The key observation for the extension is that each edge weight $\omega_{D_1}((u, v))$ in D_1 or $\omega_{D_2}((u, v))$ in D_2 of the scalable k-reach index actually encodes the shortest path distance from u to v in G. Let $d(s, t)$ be the exact shortest path distance from any vertex s to another vertex t in G. For any k, the scalable k-reach index, consisting of $D_1 = (V_{D_1}, E_{D_1}, \omega_{D_1})$, $D_2 = (V_{D_2}, E_{D_2}, \omega_{D_2})$, and a residual graph D_3, can be used to obtain $d(s, t)$, if $d(s, t) \leq k$.

With the above observation, if we construct a scalable δ-reach index by Algorithm 3, where δ is the diameter of the input graph, we can answer k-hop reachability queries with any value of k. We outline the extended query processing algorithm in Algorithm 5.

Algorithm 5 is similar to Algorithm 4. The only difference is that in Algorithm 4, if $(s, t) \in E_{D_1}$ (or $(s, t) \in E_{D_2}$), we know for sure that $\omega_{D_1}((s, t)) \leq k$ (or $\omega_{D_2}((s, t)) \leq k$) because the input is a k-reach index; while in Algorithm 5, even if we know $(s, t) \in E_{D_1}$ (or $(s, t) \in E_{D_2}$), we still need to test if $\omega_{D_1}((s, t)) \leq k$ (or $\omega_{D_2}((s, t)) \leq k$) because the input is a δ-reach index and $\delta \geq k$.

The complexity analysis of Algorithm 5 follows the same way as that of Algorithm 4 given in Sect. 5.5. In fact, the complexity remains the same because in Sect. 5.5, we give the worst case complexity, but note that D_1, D_2, and D_3 belong to a δ-reach index, which is at least as large as those of any k-reach index since $k \leq \delta$.

Finally, we note that, based on the above-mentioned observation, our index can also be used to return the exact shortest path distance from s to t. If we use a scalable k-reach index,

Algorithm 5: Query processing for general k using scalable δ-reach

```
Input  : A more scalable $\delta$-reach index of $G$ ($\delta$ is the diameter of
         $G$), consisting of $D_1 = (V_{D_1}, E_{D_1}, \omega_{D_1})$,
         $D_2 = (V_{D_2}, E_{D_2}, \omega_{D_2})$, a residual graph $D_3$, and two
         query vertices, $s$ and $t$
Output: A boolean indicator whether $s \to_k t$
   //case 1: both $s$ and $t$ are in $S$
1  if $s \in S$ and $t \in S$ then
2      if $(s,t) \in E_{D_1}$ and $\omega_{D_1}((s,t)) \leq k$ then
3          return true;
4      else
5          return false;
   //case 2: $s$ or $t$ is in $S$, but not both
6  else if $s \in S$ or $t \in S$ then
7      if $(s,t) \in E_{D_1}$ and $\omega_{D_1}((s,t)) \leq k$,
8      or $\exists v \in S$ such that $\omega_{D_1}((s,v)) + \omega_{D_1}((v,t)) \leq k$, then
9          return true;
10     else
11         return false;
   //case 3: both $s$ and $t$ are in $S'$
12 else if $s \in S'$ and $t \in S'$ then
13     if $(s,t) \in E_{D_2}$ and $\omega_{D_2}((s,t)) \leq k$,
14     or $\exists u, v \in S$ such that
       $\omega_{D_1}((s,u)) + \omega_{D_1}((u,v)) + \omega_{D_1}((v,t)) \leq k$, then
15         return true;
16     else
17         return false;
   //case 4: $s, t \notin S$ and $s$ or $t$ is in $S'$
18 else if $s \in S'$ or $t \in S'$ then
19     if $(s,t) \in E_{D_2}$ and $\omega_{D_2}((s,t)) \leq k$,
20     or $\exists v \in S'$ such that $\omega_{D_2}((s,v)) + \omega_{D_2}((v,t)) \leq k$, or
       $\exists u, v \in S$ such that
       $\omega_{D_1}((s,u)) + \omega_{D_1}((u,v)) + \omega_{D_1}((v,t)) \leq k$, then
21         return true;
22     else
23         return false;
   //case 5: both $s$ and $t$ are not $S$ or $S'$
24 else if $s, t \in V_{D_3}$ then
25     if $\exists u, v \in S$ such that
       $\omega_{D_1}((s,u)) + \omega_{D_1}((u,v)) + \omega_{D_1}((v,t)) \leq k$,
26     or $\exists u, v \in S'$ such that
       $\omega_{D_2}((s,u)) + \omega_{D_2}((u,v)) + \omega_{D_2}((v,t)) \leq k$ then
27         return true;
28     else
29         start BFS from $s$ in $D_3$ and from $t$ in the reverse graph of
           $D_3$ in parallel for $\lceil k/2 \rceil$ hops: if the two BFSs meet at a
           common vertex $v$ and $d(s,v) + d(t,v) \leq k$, return
           true; otherwise, return false;
```

then we can obtain $d(s,t)$ if $d(s,t) \leq k$, or return `false` if $d(s,t) > k$ (i.e., $s \not\to_k t$). If we use a scalable δ-reach index, then the index is simply an index for processing general shortest path distance queries. As processing general shortest path distance queries is not the focus of this paper, we do not go into the details but give the algorithm for processing general shortest path distance queries using our index in Appendix.

6 Experimental evaluation

We now evaluate the performance of our index, k-reach. Since this is the first work that proposes an index for processing k-hop reachability queries, we compare with closely related works, that is, (1) the state-of-the-art methods that were proposed for answering shortest path distance queries [3,25] and (2) the state-of-the-art methods that were proposed for processing classic reachability queries [26,27,33,37]. We will evaluate the performance of k-reach for processing both k-hop reachability queries and classic reachability queries by comparing with the performance of the closely related methods.

All systems, both ours and others we compared with, were implemented in C++ and compiled using the same gcc compiler. We ran all experiments on machines with an Intel 3.3 GHz CPU, 16GB RAM, and running Ubuntu 11.04 Linux OS. The experiments were run for 10 times, and the results were found to be consistent over the 10 runs.

6.1 Datasets

We conducted our experiments on a wide range of real datasets that are popularly used to assess the performance of graph reachability indexes in the existing works [26–28,33,37].

We group the datasets into two sets. The first set of datasets is relatively smaller datasets. The datasets `AgroCyc`, `Anthra`, `Ecoo`, `Human`, `Mtbrv`, and `Vchocyc`, are from EcoCyc (*ecocyc.org*) and describe the genome and biochemical machinery of E. coli K-12 MG1655. The `aMaze` and `Kegg` datasets are metabolic networks from [32]. The `Nasa` and `Xmark` datasets are XML documents from [28]. The datasets `ArXiv` (*arxiv.org*), `CiteSeer` (*citeseer.ist.psu.edu*), and `PubMed` (*pubmedcentral.nih.gov*), are citation networks. The `GO` dataset (*www.geneontology.org*) is a gene ontology graph. The `YAGO` dataset (*mpi-inf.mpg.de/yago-naga/yago*) is a graph that describes the structure of relationships among terms in the semantic knowledge data-base YAGO.

The second set of datasets are larger datasets. The `citeseerx` and `cit-patent` (`patent`) datasets are citation networks (*snap.stanford.edu/data*), in which non-leaf vertices have an average out-degree of 10 to 30. The `go-uniprot` dataset is the joint graph of Gene Ontologyterm and the annotations from the UniProt database (*www.uniprot.org*), which is the universal protein resource. The `uniprot22m`, `uniprot100m` and `uniprot150m` datasets are the subsets of the complete RFG graph of

UniProt. These larger datasets are mainly used in some more recent works to test the scalability of reachability indexes.

Table 3 shows the number of vertices and edges ($|V|$ and $|E|$), the maximum vertex degree (deg_{max}), the diameter (δ), and the median length of the shortest paths between all pairs of vertices (μ) of the datasets. We note that the value μ for the six larger datasets is approximated by the method proposed in [4], while it is too expensive to compute the exact value of μ and the value of δ for these large graphs.

6.2 Query set

To compare the performance of query processing of the various indexes, we randomly generated 1 million queries for each dataset. We emphasize that, as we will explain later in details in Tables 10 and 11 in Sect. 6.5, these queries are not chosen to favor the performance of our index. In fact, the majority of these queries fall into the case which is the worst case for query processing using k-reach.

6.3 Evaluation on different aspects of the scalable k-reach

Before we presented the final scalable k-reach index in Sect. 5, we first introduced (1) a one-level index constructed based on a partial vertex cover with maximum coverage of vertices (we name this index as **pre-k-reach-v1** in this experiment), and (2) a two-level index which alleviates the problem of the one-level index (we name this index as **pre-k-reach-v2** in this experiment). Since these two indexes, pre-k-reach-v1 and pre-k-reach-v2, lead to the design of the final scalable k-reach index, we examine here how the weaknesses in pre-k-reach-v1 and pre-k-reach-v2 are addressed to give our final index.

Tables 4, 5 and 6 report the index construction time, the index size, and the total query time for processing 1 million randomly generated queries (where k is set to μ, i.e., the median length of the shortest paths between all pairs of vertices in each dataset). In this experiment, we focus on the large datasets only because the more scalable k-reach index proposed in Sect. 5 is primarily designed for handling large datasets.

Table 3 Datasets

	$\|V\|$	$\|E\|$	deg_{max}	δ	μ
AgroCyc	12,684	13,657	5,488	10	2
aMaze	3,710	3,947	3,097	11	2
Anthra	12,499	13,327	5,401	10	2
ArXiv	6,000	66,707	700	20	5
CiteSeer	10,720	44,258	192	18	3
Ecoo	12,620	13,575	5,435	10	2
GO	6,793	13,361	71	11	3
Human	38,811	39,816	28,571	10	3
Kegg	3,617	4,395	3,282	16	2
Mtbrv	9,602	10,438	4,005	12	2
Nasa	5,605	6,538	32	22	7
PubMed	9,000	40,028	432	11	3
Vchocyc	9,491	10,345	3,917	10	2
Xmark	6,080	7,051	887	24	5
YAGO	6,642	42,392	2,371	9	1
citeseerx	6,540,401	15,011,260	384,942	–	4
go-uniprot	6,967,956	34,770,235	1,186,282	–	4
patent	3,774,768	16,518,947	793	–	8
uniprot22m	1,595,444	1,595,442	1,539,898	–	2
uniprot100m	16,087,295	16,087,293	12,875,285	–	2
uniprot150m	25,037,600	25,037,598	17,661,135	–	2

$|V|$ the number of vertices; $|E|$ the number of edges; deg_{max} the maximum vertex degree; δ the diameter; μ the median length of all shortest paths in the graph

Table 4 Index construction time (elapsed time in ms) of pre-k-reach-v1, pre-k-reach-v2, and k-reach presented in Sect. 5

	pre-k-reach-v1	pre-k-reach-v2	k-reach
citeseerx	–	32,663.86	**1,687.33**
go-uniprot	6,493.60	3,775.40	**3,366.05**
patent	60,793.11	4,355.52	**2,578.73**
uniprot22m	230.22	**113.77**	249.18
uniprot100m	888.00	**846.88**	12,785.97
uniprot150m	**1,314.95**	1,432.66	18,973.30

Shortest time shown in bold

Table 5 Index size (in MB) of pre-k-reach-v1, pre-k-reach-v2, and k-reach presented in Sect. 5

	pre-k-reach-v1	pre-k-reach-v2	k-reach
citeseerx	–	1,351.68	**61.71**
go-uniprot	468.92	354.12	**114.19**
patent	2,337.11	198.21	**62.05**
uniprot22m	**12.70**	16.80	16.74
uniprot100m	**126.84**	169.59	167.78
uniprot150m	**196.43**	263.49	259.14

Smallest size shown in bold

Table 6 Total query time (elapsed time in ms) of pre-k-reach-v1, pre-k-reach-v2, and k-reach presented in Sect. 5 for processing 1 million randomly generated queries

	pre-k-reach-v1	pre-k-reach-v2	k-reach
citeseerx	–	282,684.09	**1,497.28**
go-uniprot	2,554.72	1,862.08	**613.90**
patent	107,085.34	**66,183.42**	69,579.86
uniprot22m	456.50	446.61	**256.70**
uniprot100m	641.38	592.37	**378.30**
uniprot150m	663.86	604.89	**430.18**

Shortest time shown in bold

The results show that for processing the `citeseerx`, `go-uniprot`, and `patent` datasets, *k*-reach is significantly more efficient than pre-*k*-reach-v1 and pre-*k*-reach-v2 in terms of both indexing and query performance. In processing the three `uniprot` datasets, *k*-reach is more expensive in indexing but more efficient in query processing, but we emphasize that online query processing is the more important performance indicator than offline indexing performance. Thus, the overall performance of *k*-reach clearly justifies why it is chosen to be our final index instead of pre-*k*-reach-v1 and pre-*k*-reach-v2.

The superiority of *k*-reach in query performance can be explained by the greater coverage by the two-level *k*-relaxed partial vertex cover employed by *k*-reach, as compared with the partial vertex cover employed by pre-*k*-reach-v1 and pre-*k*-reach-v2, although this comes at an expense of higher indexing cost for some datasets. Compared with *k*-reach and pre-*k*-reach-v2 which employ a two-level index structure, we can see from the results that the one-level index, pre-*k*-reach-v1, is significantly less efficient in query processing.

We note that the query performance of the indexes does not vary significantly for different budget values. The explanation for this is because the majority of the queries belong to the case which is the worst case for query processing using *k*-reach (see details in Sect. 6.5). We could easily generate the queries so that they are evenly distributed to the five cases as presented in Algorithm 4, but then, it is expected that query performance becomes proportionally better as larger budget includes more queries into the first four cases. However, such a result is expected, while in real applications queries are certainly not generated in this way. Thus, we stick to using the randomly generated queries in our experiments. For the results presented in this subsection and subsequent discussion, we use a budget $b = 1000$.

6.4 Performance of processing *k*-hop reachability queries

In this experiment, we evaluate the performance of the *k*-reach index for processing *k*-hop reachability queries. Since *k*-reach is the first index for processing *k*-hop reachability queries, we compare with the state-of-the-art indexes for processing shortest path distance queries in directed graphs, namely the *highway-centric labeling* approach (denoted by **HCL**) [25] and the *pruned landmark labeling* approach (denoted by **PLL**) [3]. We also compare with the (h, k)-reach index, which is a method used to reduce the index size at the expense of query performance [11].

We report the index construction time, the index size, and the total query time for processing 1 million randomly generated queries in Tables 7, 8 and 9. We set $k = \mu$ for both *k*-reach and (h, k)-reach, while $h = 2$ for (h, k)-reach which is the best value of h reported in [11].

Table 7 Index construction time (elapsed time in ms) of *k*-reach, (*h*,*k*)-reach, HCL, and PLL

	k-reach	(*h*,*k*)-reach	HCL	PLL
`AgroCyc`	22.67	30.94	12,628.83	**21.14**
`aMaze`	13.93	12.06	26,835.21	**8.28**
`Anthra`	**20.37**	31.45	12,660.50	21.22
`ArXiv`	183.45	831.22	209,056.20	**66.62**
`CiteSeer`	178.77	676.38	18,016.52	**98.98**
`Ecoo`	**22.12**	34.06	12,523.01	22.13
`GO`	80.87	152.35	5,020.84	**35.45**
`Human`	**49.03**	72.25	108,930.20	79.12
`Kegg`	16.31	14.67	31,331.48	**7.12**
`Mtbrv`	19.54	25.33	7,591.70	**17.51**
`Nasa`	47.04	59.54	4,250.49	**27.44**
`PubMed`	119.34	673.10	17,352.12	**63.57**
`Vchocyc`	19.10	25.22	7,380.21	**17.36**
`Xmark`	33.57	54.33	9,431.89	**14.42**
`YAGO`	**23.14**	176.71	4,455.69	28.38
`citeseerx`	**1687.33**	–	–	–
`go-uniprot`	**3,366.05**	–	–	161,328.70
`patent`	**2,578.73**	–	–	–
`uniprot22m`	**249.18**	–	–	6,326.11
`uniprot100m`	**12,785.97**	–	–	–
`uniprot150m`	**18,973.30**	–	–	–

Shortest time shown in bold

Table 8 Index size (in MB) of *k*-reach, (*h*,*k*)-reach, HCL, and PLL

	k-reach	(*h*,*k*)-reach	HCL	PLL
`AgroCyc`	0.14	**0.04**	0.80	10.45
`aMaze`	0.08	**0.03**	0.16	2.76
`Anthra`	0.13	**0.03**	0.78	10.29
`ArXiv`	3.88	4.52	**2.32**	5.15
`CiteSeer`	5.45	5.81	**0.75**	9.49
`Ecoo`	0.14	**0.04**	0.79	10.40
`GO`	2.19	2.34	**0.55**	5.88
`Human`	0.34	**0.04**	2.48	31.98
`Kegg`	0.12	**0.05**	0.21	2.98
`Mtbrv`	0.11	**0.03**	0.61	7.91
`Nasa`	0.86	0.57	**0.55**	5.02
`PubMed`	3.11	3.14	**0.68**	7.72
`Vchocyc`	0.11	**0.03**	0.60	7.82
`Xmark`	**0.41**	0.44	0.60	5.03
`YAGO`	0.35	**0.21**	0.43	5.51
`citeseerx`	**61.71**	–	–	–
`go-uniprot`	**114.19**	–	–	5,941.60
`patent`	**62.05**	–	–	–
`uniprot22m`	**16.74**	–	–	1,314.83
`uniprot100m`	**167.78**	–	–	–
`uniprot150m`	**259.14**	–	–	–

Smallest size shown in bold

Table 9 Total query time (elapsed time in ms) of *k*-reach, (*h*,*k*)-reach, HCL, and PLL, for processing 1 million randomly generated queries

	k-reach	(*h*,*k*)-reach	HCL	PLL
AgroCyc	**5.29**	56.02	13.61	624.62
aMaze	**13.20**	634.71	63.26	491.37
Anthra	**5.13**	54.52	13.06	588.02
ArXiv	**43.62**	297.64	13,338.44	737.43
CiteSeer	**43.89**	201.48	367.65	880.96
Ecoo	**5.27**	56.24	13.69	611.37
GO	**24.30**	60.90	119.76	736.10
Human	**6.50**	52.03	12.69	599.73
Kegg	**14.94**	682.13	79.18	492.51
Mtbrv	**5.15**	56.61	14.08	611.90
Nasa	**14.45**	53.93	54.38	715.16
PubMed	**33.72**	128.14	339.65	797.66
Vchocyc	**5.08**	57.71	14.00	596.66
Xmark	**11.80**	69.35	53.48	582.77
YAGO	**25.38**	212.14	161.54	659.67
citeseerx	**1,497.28**	–	–	–
go-uniprot	**613.90**	–	–	977.53
patent	**69,579.86**	–	–	–
uniprot22m	**256.70**	–	–	834.38
uniprot100m	**378.30**	–	–	–
uniprot150m	**430.18**	–	–	–

Shortest time shown in bold

For the small datasets, PLL has the shortest indexing time for most of the datasets, but *k*-reach's indexing time is not much longer than PLL's. HCL is clearly too expensive to construct even for these small datasets. For query processing, *k*-reach is clearly the winner in all cases, as it is from a few times to a hundred times faster than HCL and at least an order of magnitude faster than PLL. Compared with (h, k)-reach, *k*-reach has a slightly larger index size but is significantly faster in indexing and query processing, showing that (h, k)-reach cannot address the scalability problem of the *k*-reach index proposed in [11], and hence, the more scalable index proposed in this paper is needed.

For processing the large datasets, our adaptive scheme (presented in Sect. 5.6) chooses the more scalable *k*-reach index proposed in Sect. 5.4. The results reported in Tables 7, 8 and 9 clearly show that *k*-reach is more scalable than (h, k)-reach, HCL, and PLL, which either ran out of the main memory capacity (16 GB) of our machine or took unreasonably long running time (longer than two orders of magnitude than that of our method) for most of the large datasets. Although PLL obtains the results for two of the six large datasets, its indexing performance is drastically worse than that of *k*-reach in terms of both indexing time and index size, while its query performance is also considerably worse than that of *k*-reach.

In conclusion, the results verify that *k*-reach is a good index for processing *k*-hop reachability queries, as evidenced by its significantly better query performance and scalability. It demonstrates the need of an index for processing *k*-hop reachability queries because the results also show that the state-of-the-art indexes for processing shortest path distance queries are not efficient and scalable enough for processing *k*-hop reachability queries, while in Sect. 3, we have shown that it is non-trivial to adopt existing classic reachability indexes for processing *k*-hop reachability queries.

In the following subsection, we further show that the randomly generated queries tested in the experiment do not favor the performance of the *k*-reach index.

6.5 The different cases of query processing

In Algorithm 2, we show that there are four cases in processing a *k*-hop reachability query, while in Algorithm 4, there are five cases. The complexity analysis in Sect. 4.2.2 and in Sect. 5.5 show that processing a Case 1 query has the lowest time complexity, and processing a Case 4 or Case 5 query is the most costly for Algorithm 2 and Algorithm 4, respectively. Thus, we want to examine whether the queries used in the experiments may favor the performance of *k*-reach.

Tables 10 and 11 present the percentage of queries in each of the different cases in Algorithm 2 and in Algorithm 4, for the 1 million queries tested.

For the small datasets, Table 10 shows that for most of the datasets, the majority of the queries are Case 4 queries, while the minority are Case 1 queries. For the large datasets, Table 11 shows that over 99 % of the queries fall into Case 5, while most of the remaining queries are also in Case 4

Table 10 Percentage of queries (among the 1 million randomly generated queries) in each of the four cases of Algorithm 2

	Case 1	Case 2	Case 3	Case 4
AgroCyc	0.10	2.98	2.96	93.97
aMaze	1.65	11.19	11.23	75.93
Anthra	0.08	2.73	2.79	94.40
ArXiv	41.94	22.79	22.88	12.38
CiteSeer	19.15	24.62	24.62	31.61
Ecoo	0.10	3.02	3.05	93.83
GO	19.18	24.63	24.66	31.53
Human	0.01	0.94	0.96	98.09
Kegg	2.92	14.17	14.21	68.71
Mtbrv	0.15	3.66	3.67	92.52
Nasa	10.80	22.12	22.03	45.05
PubMed	15.12	23.77	23.71	37.40
Vchocyc	0.15	3.65	3.68	92.53
Xmark	4.06	16.08	16.10	63.75
YAGO	1.55	10.96	10.89	76.60

Table 11 Percentage of queries (among the 1 million randomly generated queries) in each of the 5 cases of Algorithm 4

	Case 1	Case 2	Case 3	Case 4	Case 5
`citeseerx`	0	0.02	0.0003	0.24	99.74
`go-uniprot`	0	0.04	0.0002	0.22	99.74
`patent`	0	0.01	0.0003	0.43	99.56
`uniprot22m`	0	0.004	0.002	0.99	99.01
`uniprot100m`	0	0.006	0.0001	0.1	99.89
`uniprot150m`	0	0.004	0	0.06	99.94

(the case with the second highest query complexity). This distribution is mainly because the size of the vertex cover or the k-relaxed partial vertex cover is only a small percentage of the total number of vertices in the graphs, and hence, a randomly selected vertex has a lower probability being in the vertex cover or the k-relaxed partial vertex cover.

The above results show that, if we select the query vertices so that they fall into the different cases evenly, the average query processing time can be significantly shorter than the results currently reported. Note that such a query selection can be done by selecting query vertices with a probability proportional to their degree; however, for fairness of comparison with other cases, we keep the query set as the randomly selected queries in the performance evaluation.

We remark that, even though up to 99% of the queries fall into the worst case of k-reach, k-reach is still an efficient and scalable index for processing k-hop reachability queries, as we have shown that k-reach is significantly more efficient and scalable than other indexes (in Sect. 6.4) and BFS (in [11]).

6.6 Performance on different values of k

We next examine the performance of k-reach for different values of k. We test $k \in \{2, 4, 6, 8, 10, 12, \mu, n\}$, but note that for some datasets and some $k = k'$, if $\delta < k'$, then the performance result of k-reach is the same as $k = n$ (e.g., $k = 10$ is the same as $k = n$ if $\delta = 9 < 10$). We use the same set of randomly generated queries for all values of k. Note that for the same query vertices, s and t, the answer whether t is reachable from s within k hops may be different for different values of k.

We found that our adaptive scheme (presented in Sect. 5.6) chooses the basic k-reach index proposed in Sect. 4 when processing the small datasets, and it chooses the more scalable version proposed in Sect. 5.4 for processing the six large datasets. Thus, we report the results for the small and large datasets separately.

6.6.1 Performance on small datasets

We first discuss the performance results for the small datasets, obtained by the basic k-reach index proposed in Sect. 4. Table 12 reports the total running time of processing the 1 million queries by k-reach, for different values of k. We omit the results of indexing performance because both the construction time and the index size of k-reach are almost the same for the different values of k; thus, we refer the readers to the results reported for k-reach (where $k = \mu$) in Tables 7 and 8.

The results in Table 12 show that the performance of the different k-reach indexes is stable with the different values of k for all the small datasets. Note that the values of k rang-

Table 12 Total query time (elapsed time in ms) of k-reach, where $k \in \{2, 4, 6, 8, 10, 12, \mu, n\}$, for processing 1 million randomly generated queries, for the small graphs

	2-reach	4-reach	6-reach	8-reach	10-reach	12-reach	μ-reach	n-reach
`AgroCyc`	5.31	5.31	5.33	5.27	5.23	5.27	5.29	5.28
`aMaze`	13.24	13.31	13.32	13.29	13.29	13.31	13.20	13.30
`Anthra`	5.14	5.13	5.09	5.11	5.09	5.11	5.13	5.09
`ArXiv`	44.70	44.65	43.61	43.36	42.69	42.59	43.62	42.67
`CiteSeer`	43.82	43.97	43.63	43.44	43.75	43.31	43.89	43.31
`Ecoo`	5.32	5.29	5.26	5.24	5.26	5.26	5.27	5.24
`GO`	24.25	24.37	24.17	24.34	24.19	24.06	24.30	24.01
`Human`	6.52	6.48	6.47	6.44	6.45	6.44	6.50	6.44
`Kegg`	14.97	15.18	15.24	15.20	15.22	15.25	14.94	15.22
`Mtbrv`	5.13	5.13	5.12	5.09	5.07	5.09	5.15	5.07
`Nasa`	14.42	14.52	14.46	14.50	14.52	14.47	14.45	14.50
`PubMed`	33.88	33.79	33.66	33.73	33.45	33.46	33.72	33.52
`Vchocyc`	5.10	5.09	5.12	5.09	5.08	5.07	5.08	5.07
`Xmark`	11.66	11.82	11.80	11.83	11.84	11.83	11.80	11.89
`YAGO`	25.42	25.44	25.36	25.31	25.26	25.36	25.38	25.35

ing from 2 to n cover the two extreme ends ($k = 1$ is trivial since it only needs to check edge existence). Thus, the result demonstrates the efficiency of the k-reach index for processing k-hop reachability queries of any k for the small datasets.

6.6.2 Performance on large datasets

Next, we present the performance results for the large datasets, obtained by the more scalable k-reach index proposed in Sect. 5.4. Tables 13 to 15 report the index construction time, the index size, and the total query time for processing 1 million randomly generated queries for the different k-reach indexes.

The results show that both indexing performance and query performance degrade, though gracefully, when the value of k increases. This is expected since the scalable k-reach index is used for processing these large datasets. Hence, the larger the value of k, the larger is the number of vertices covered, and thus, the larger is the size of the index and the index construction cost. The query processing time largely depends on the index size and hence also increases as k increases.

We notice a surprising phenomenon for indexing the datasets `uniprot100m` and `uniprot150m`, however, as the indexing time first increases from $k = 2$ to $k = 4$ and then drops significantly starting $k = 6$, and gradually becomes stable. Note that $\mu = 2$ for these two datasets, and hence, μ-reach is in fact 2-reach, and the small difference in the results reported for μ-reach and 2-reach is just because the results were collected from different runs of the same program. We examined the two datasets closely and found that when $k > 4$, most of the vertices in the graphs are covered quickly by the first-level index, and hence, a significantly less number of BFS is executed. The results also mean that the performance of k-reach reported in Sect. 6.4 is actually the worst performance of k-reach for any k for these two datasets.

Table 13 Index construction time (elapsed time in ms) of k-reach, where $k \in \{2, 4, 6, 8, 10, 12, \mu, n\}$, for the large graphs

	2-reach	4-reach	6-reach	8-reach	10-reach	12-reach	μ-reach	n-reach
`citeseerx`	1,204.12	1,680.79	1,705.64	1,857.45	1,662.18	1,657.89	1,687.33	1,753.23
`go-uniprot`	2,521.51	3,368.41	3,513.85	3,374.41	3,347.57	3,382.76	3,366.05	3,443.61
`patent`	647.48	1,379.63	2,180.24	2,594.61	2,778.38	2,722.73	2,578.73	2,749.48
`uniprot22m`	248.48	234.15	234.03	234.33	233.99	234.03	249.18	234.06
`uniprot100m`	12,801.62	14,682.78	3,190.21	2,916.94	2,915.32	2,914.77	12,785.97	2,917.02
`uniprot150m`	18,936.10	20,081.49	7,744.62	4,868.32	4,885.96	4,878.05	18,973.30	4,875.51

Table 14 Index size (in MB) of k-reach, where $k \in \{2, 4, 6, 8, 10, 12, \mu, n\}$, for the large graphs

	2-reach	4-reach	6-reach	8-reach	10-reach	12-reach	μ-reach	n-reach
`citeseerx`	61.40	61.71	62.10	62.15	62.83	62.74	61.71	62.88
`go-uniprot`	116.39	114.19	119.74	118.67	118.48	122.12	114.19	122.18
`patent`	43.43	49.22	57.43	62.05	64.95	64.22	62.05	64.50
`uniprot22m`	16.74	16.74	16.74	16.74	16.74	16.74	16.74	16.74
`uniprot100m`	167.78	168.79	168.76	168.76	168.76	168.76	167.78	168.76
`uniprot150m`	259.14	262.59	262.66	262.65	262.65	262.65	259.14	262.65

Table 15 Total query time (elapsed time in ms) of k-reach, where $k \in \{2, 4, 6, 8, 10, 12, \mu, n\}$, for processing 1 million randomly generated queries, for the large graphs

	2-reach	4-reach	6-reach	8-reach	10-reach	12-reach	μ-reach	n-reach
`citeseerx`	472.75	1,520.68	3,613.43	5,905.23	7,535.72	8,444.22	1,497.28	9,102.79
`go-uniprot`	497.85	619.40	676.77	676.94	672.12	684.62	613.90	680.75
`patent`	3,137.97	18,477.71	47,288.82	69,789.80	77,827.87	79,825.51	69,579.86	80,395.65
`uniprot22m`	247.13	246.90	249.10	251.11	247.90	251.47	256.70	249.05
`uniprot100m`	379.41	378.12	380.10	378.42	382.28	380.99	378.30	398.77
`uniprot150m`	431.04	444.92	434.67	438.85	445.42	432.91	430.18	439.01

6.7 Vertex degree of the k-reach graphs

From the complexity analysis of Algorithms 2 and 4 shown in Sects. 4.2.2 and 5.5, we can see that the complexity of query processing depends largely on the degree (i.e., the number of in-neighbors or out-neighbors) of the vertices in the k-reach graphs I, or D_1 and D_2. Thus, we show the average and maximum vertex degree in these k-reach graphs (where k is set to μ) so that we may have a better understanding of the query processing algorithms and their complexity.

Table 16 reports the average and maximum degree (in-degree or out-degree) of the vertices in the k-reach graph I (defined in Sect. 4.1.2), while Table 17 reports the average and maximum degree of the vertices in the k-reach graphs D_1 and D_2 (defined in Sect. 5). The results show that, although the maximum vertex degree can be quite large for many datasets, the average vertex degree is actually very small (less than 1% of the total number of vertices for most of the datasets), indicating that the query complexity is small in the average case.

The results also show that both the average and maximum vertex degree in D_2 are significantly smaller than those in D_1, which is because the high-degree vertices are included in D_1 and removed from the input graph before we compute D_2.

Table 16 Average and maximum vertex degree in the k-reach graph I, where $k = \mu$

	Average degree	Maximum degree
AgroCyc	93	5,761
aMaze	361	1,717
Anthra	75	5,499
ArXiv	725	3,178
CiteSeer	25	345
Ecoo	92	5,708
GO	16	1,134
Human	309	28,604
Kegg	292	1,828
Mtbrv	74	4,260
Nasa	36	3,362
PubMed	57	614
Vchocyc	79	4,174
Xmark	126	4,403
YAGO	6	56

Table 17 Average and maximum vertex degree in the k-reach graphs D_1 and D_2, where $k = \mu$

	Average degree (D_1)	Maximum degree (D_1)	Average degree (D_2)	Maximum degree (D_2)
citeseerx	526	348,750	126	9,539
go-uniprot	6,008	3,242,101	23	11,306
patent	2,792	175,176	896	65,564
uniprot22m	53,183	1,595,236	0	0
uniprot100m	7,859	15,551,205	12	57
uniprot150m	11,883	23,415,904	21	122

6.8 Performance of k-reach for general k

In Sects. 4.4 and 5.7, we discuss how k-reach may be extended to handle different values of k. Note that this is different from the experiment presented in Sect. 6.6, in which a different k-reach index is constructed for each specific value of k. Here, we evaluate the performance of a single index for processing k-hop reachability queries for any k.

For the basic k-reach index, we generalize it to process shortest path distance queries as described in Sects. 4.4, and we name it as ***k*-dist** in this experiment. We report the index construction time, the index size, and the total query time for processing 1 million randomly generated queries in Table 18. We also report the results of k-reach (where k is set to μ) for reference. The result shows that both the indexing time and query time of k-dist are comparable with those of k-reach. This is because the algorithms for index construction and query processing of k-dist are basically the same for k-dist as for k-reach. Thus, if the k-reach graph covers most of the vertices in the input graph, the only big difference is that k-dist uses $\lg \delta$ bits for each edge weight (instead of 2 bits in k-reach), where δ is the diameter of the input graph. Thus, we can see that k-dist is on average 2.42 times larger than k-reach.

For the more scalable version of k-reach, we extend it by Algorithm 5 (denoted by **gen-*k*-reach**) for processing general k-hop reachability queries, and by Algorithm 6 in Appendix (denoted by ***k*-dist**) for processing shortest path distance queries, as shown in Table 19. We report the total query time only because, as discussed in Sect. 5.7, the indexing time and index size of both gen-k-reach and k-dist are exactly the same as those of n-reach, which are already reported in Tables 13 and 14. We also report the query time of k-reach (where k is set to μ) for reference.

The result shows that for most datasets, the query time of gen-k-reach and k-dist is not much longer than that of k-reach, which is not surprising since the query complexity of the three algorithms are actually the same. In general, gen-k-reach and k-dist take longer time to process a query than k-reach. However, compared with the results of the state-of-the-art indexes for processing shortest path distance queries, HCL [25] and PLL [3], as shown in Table 9, gen-k-reach and k-dist are significantly more efficient and scalable.

This result demonstrates the flexibility of our indexing technique even for processing shortest path distance queries.

Table 18 Performance of k-dist for processing general k and shortest path distance queries, with reference to k-reach for processing μ-hop reachability queries, on the small datasets

	Indexing time (ms) (***k*-dist**)	Indexing time (ms) (***k*-reach**)	Index size (MB) (***k*-dist**)	Index size (MB) (***k*-reach**)	Query time (ms) (***k*-dist**)	Query time (ms) (***k*-reach**)
AgroCyc	21.30	22.67	0.16	0.14	6.00	5.29
aMaze	14.15	13.93	0.22	0.08	13.70	13.20
Anthra	18.15	20.37	0.13	0.13	5.80	5.13
ArXiv	198.45	183.45	14.41	3.88	75.93	43.62
CiteSeer	176.68	178.77	20.97	5.45	55.80	43.89
Ecoo	23.14	22.13	0.16	0.14	6.00	5.27
GO	83.64	80.87	8.46	2.19	33.53	24.30
Human	47.00	49.03	0.17	0.34	6.73	6.50
Kegg	19.26	16.31	0.37	0.12	15.53	14.94
Mtbrv	15.52	19.54	0.14	0.11	5.99	5.15
Nasa	46.67	47.04	3.24	0.86	22.12	14.45
PubMed	119.06	119.34	11.68	3.11	47.24	33.72
Vchocyc	15.16	19.10	0.13	0.11	5.99	5.08
Xmark	34.59	33.57	1.43	0.41	16.68	11.80
YAGO	23.80	23.14	0.67	0.35	30.38	25.38

Table 19 Total query time (elapsed time in ms) of gen-k-reach for general k and k-dist for processing shortest path distance queries, with reference to k-reach for processing μ-hop reachability queries, on the large datasets

	gen-k-reach	k-dist	k-reach
citeseerx	9,200.29	9,699.34	1,497.28
go-uniprot	693.70	667.29	613.90
patent	80,425.12	82,228.21	69,579.86
uniprot22m	244.55	293.58	256.70
uniprot100m	345.20	433.50	378.30
uniprot150m	402.34	484.70	430.18

In the following subsection, we further show that our index is also efficient for processing classic reachability queries.

6.9 Performance of processing classic reachability queries

In this experiment, we report the performance of k-reach for processing classic reachability queries, by setting $k = n$ or $k = \infty$. We want to demonstrate that, although processing k-hop reachability queries has higher cost than processing classic reachability queries since some distance information needs to be handled for k-hop reachability, k-reach is also a reasonably good index for processing even classic reachability queries.

We compare with the state-of-the-art indexes for processing classic reachability queries, which include *path-tree cover* (**PTree**) [26], **3-hop**[1] [27], **GRAIL** [37], and *Partitioned Word Aligned Hybrid compression* (**PWAH**) [33]. We denote our index as ***n*-reach**, as to indicate that this is for the case when $k = n$, which is essentially an index for processing reachability queries.

6.9.1 Performance on index construction

Table 20 reports the index construction time of all the indexes for all the datasets.

For the smaller datasets, the results show that constructing the n-reach index is faster than constructing the PTree index in most cases. Compared with GRAIL and PWAH, however, constructing n-reach is considerably slower. For processing the large datasets, the results show that constructing the n-reach index is the most efficient in all cases. On average, constructing n-reach is 6.36 and 50.22 times faster than constructing GRAIL and PWAH. We were not able to obtain the results for most of the large datasets for PTree and 3-hop, since they either ran out of the main memory capacity (16GB) of our machine or took unreasonably long running time (longer than two orders of magnitude than that of our method).

Table 21 reports the storage size on disk (including all data structures, also the input graph if needed, that are used for query processing) of the various indexes for the different datasets. The results show that the size of PWAH is the smallest for most of datasets. The size of n-reach is comparable with that of PTree and GRAIL but considerably larger than that of PWAH for the small datasets. However, for the large datasets, the size of n-reach is the smallest or among the smallest. We also remark that our index is mainly designed

[1] Note that *3-hop* is only the name of the index [27] for processing classic reachability queries, and does not imply 3-hop reachability.

Table 20 Index construction time (elapsed time in ms) of *n*-reach, PTree, 3-hop, GRAIL, and PWAH

	n-reach	PTree	3-hop	GRAIL	PWAH
AgroCyc	22.40	54.31	18,565.46	5.75	**1.75**
aMaze	14.21	236.46	462,194.20	**1.72**	2.46
Anthra	19.61	51.44	18,774.03	5.55	**1.63**
ArXiv	214.18	5,974.43	–	**5.72**	67.85
CiteSeer	180.31	215.29	19,101.36	**7.97**	51.44
Ecoo	22.81	54.45	18,196.82	5.71	**1.77**
GO	81.81	63.28	3,978.72	**3.51**	8.11
Human	50.17	174.09	–	27.72	**3.15**
Kegg	16.30	271.78	–	**1.70**	2.90
Mtbrv	17.30	41.17	10,780.23	4.03	**1.63**
Nasa	45.82	29.78	5,452.37	**2.45**	4.01
PubMed	120.62	275.27	32,461.93	**6.25**	42.13
Vchocyc	18.32	40.65	10,727.72	4.07	**1.74**
Xmark	35.21	63.61	56,348.11	**2.70**	4.86
YAGO	23.30	109.69	4,904.51	**5.14**	14.48
citeseerx	**1,753.23**	–	–	7,331.45	14,191.77
go-uniprot	**3,443.61**	–	–	13,942.16	24,548.55
patent	**2,749.48**	–	–	6,635.41	751,736.47
uniprot22m	**234.06**	12,420.24	–	2,050.53	1,033.37
uniprot100m	**2,917.02**	–	–	27,826.83	12,074.84
uniprot150m	**4,875.51**	–	–	45,018.57	20,042.67

Shortest time shown in bold

Table 21 Index size (in MB) of *n*-reach, PTree, 3-hop, GRAIL, and PWAH

	n-reach	PTree	3-hop	GRAIL	PWAH
AgroCyc	0.14	0.65	0.38	0.29	**0.06**
aMaze	0.08	0.19	5.44	0.09	**0.06**
Anthra	0.13	0.64	0.22	0.29	**0.06**
ArXiv	3.88	2.58	–	0.37	**0.31**
CiteSeer	5.45	1.39	**0.29**	0.37	0.39
Ecoo	**0.14**	0.64	0.37	0.29	0.06
GO	2.19	0.51	0.17	0.18	**0.08**
Human	0.34	1.94	–	0.89	**0.16**
Kegg	0.12	0.19	–	0.09	**0.07**
Mtbrv	0.11	0.49	0.28	0.22	**0.05**
Nasa	0.86	0.30	0.11	0.13	**0.06**
PubMed	3.11	1.36	0.36	**0.32**	0.35
Vchocyc	0.11	0.49	0.29	0.22	**0.05**
Xmark	0.41	0.32	0.48	0.14	**0.06**
YAGO	0.35	1.08	0.14	0.29	**0.12**
citeseerx	**62.88**	–	–	99.83	148.78
go-uniprot	122.18	–	–	**106.42**	242.66
patent	64.50	–	–	**57.70**	5,334.12
uniprot22m	16.74	**6.13**	–	24.44	18.64
uniprot100m	**168.76**	–	–	245.75	208.63
uniprot150m	**262.65**	–	–	382.25	349.25

Smallest size shown in bold

Table 22 Total query time (elapsed time in ms) of n-reach, PTree, 3-hop, GRAIL, and PWAH, for processing 1 million randomly generated queries

	n-reach	PTree	3-hop	GRAIL	PWAH
AgroCyc	**5.28**	28.28	571.94	68.83	11.81
aMaze	**13.30**	37.52	14,032.12	2,612.58	22.84
Anthra	**5.09**	27.45	287.34	61.48	11.31
ArXiv	**42.67**	3,067.89	–	1,772.53	216.52
CiteSeer	**43.31**	337.45	653.69	169.88	238.09
Ecoo	**5.24**	28.10	534.00	74.16	11.64
GO	**24.01**	150.59	269.22	95.37	40.89
Human	**6.44**	32.47	–	203.35	8.86
Kegg	**15.22**	40.75	–	3,600.05	22.26
Mtbrv	**5.07**	27.10	557.63	63.68	12.49
Nasa	**14.50**	60.02	223.89	58.81	33.07
PubMed	**33.52**	547.93	707.32	174.82	260.74
Vchocyc	**5.07**	27.14	549.58	61.71	12.69
Xmark	**11.89**	41.22	297.59	215.43	90.44
YAGO	**25.35**	149.00	389.48	90.94	93.17
citeseerx	9,102.79	–	–	436.93	**180.29**
go-uniprot	680.75	–	–	**82.41**	438.57
patent	80,395.65	–	–	**8,859.53**	14,450.54
uniprot22m	249.05	428.54	–	**78.57**	223.76
uniprot100m	398.77	–	–	**165.27**	287.82
uniprot150m	439.01	–	–	**207.04**	300.83

Shortest time shown in bold

for processing k-hop reachability queries, and therefore, it is reasonable that it uses more space since more distance information is required to be indexed.

6.9.2 *Performance of query processing*

Table 22 reports the total time used to process the 1 million queries by the different indexes. The results show that query processing by the n-reach index is significantly faster than all the other indexes for processing the small datasets. However, for processing the larger datasets, n-reach is slower than GRAIL and PWAH, especially for the datasets `citeseerx` and `patent`. The main reason for this is because for processing the large datasets, the mechanism in processing n-hop reachability queries requires joins and complete BFS in D_3, which has significantly high complexity than the mechanism employed by GRAIL and PWAH for processing classic reachability queries. Overall, the results show that n-reach is still a reasonably good index even for processing classic reachability queries, especially given the fact that it is primarily designed for processing k-hop reachability queries.

6.9.3 *An examination on the remarkably short query time*

As shown in Table 22, the query time of n-reach for processing 1 million queries is only 5 ms for a number of datasets, which gives the remarkably short average query time of only 5 nanoseconds per query. As shown in Table 21, the index sizes of the corresponding datasets (all of them are small datasets) are all so small that they are in fact smaller than the size of the L3 cache (6 MB) and even the L2 cache (256 KB). Thus, one may question whether the remarkably short average query time is because the index is being kept in the cache memory. Since such short query time appears only on the small datasets, we want to see how the query time may change if we gradually increase the dataset size. In particular, we examine whether the fast query response time is due to some caching effect. To do this, we test the performance of our index on a set of synthetic datasets with varying sizes. We generate synthetic datasets that follow a power-law degree distribution by the model proposed in [18]. We set the average degree to 10 and vary the number of vertices from 1 to 200 K.

Figures 3 and 4 report the index construction time (elapsed time in ms) and index size (in MB) of n-reach for processing synthetic datasets with $1K \leq |V| \leq 200\,$K. We also randomly generate 1 million queries for each of the synthetic datasets and report the total query time (elapsed time in ms) of n-reach in Fig. 5.

The results show that both the indexing time and the index size increase slowly as the dataset size increases. However, Fig. 5 shows that there is a much more rapid increase in the query time when $|V|$ increases from 1 to 5 K and then from 5 to 10 K, and the query time increases only slowly after

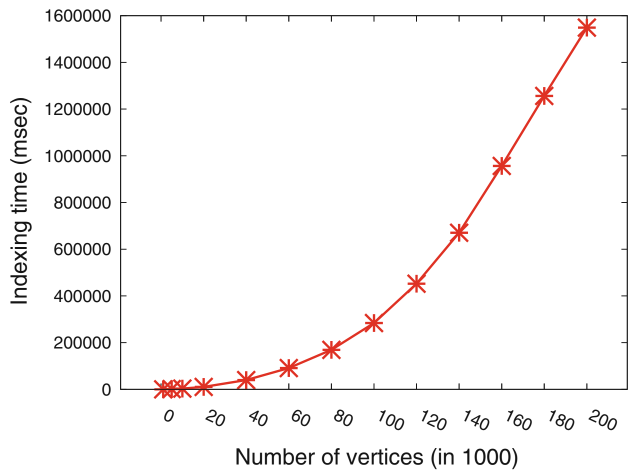


Fig. 3 Index construction time (elapsed time in ms) of n-reach for processing synthetic datasets with $1K \leq |V| \leq 200$ K

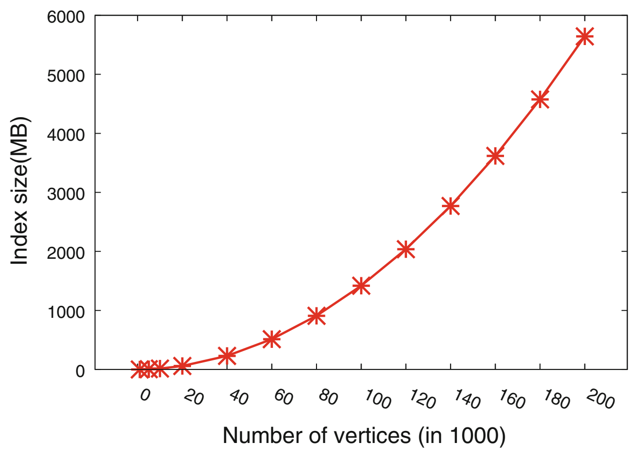


Fig. 4 Index size (in MB) of n-reach for processing synthetic datasets with $1K \leq |V| \leq 200$ K

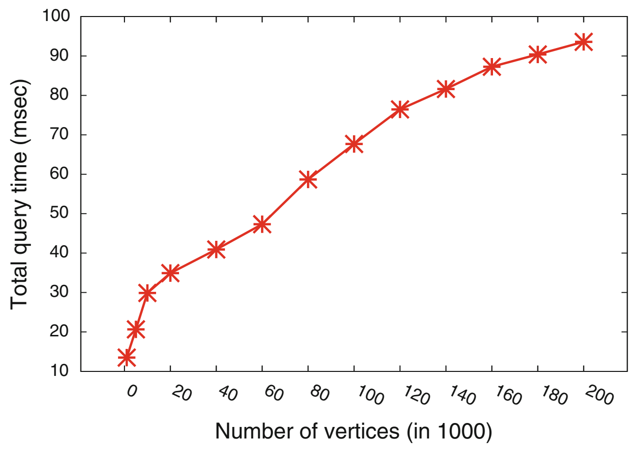


Fig. 5 Total query time (elapsed time in ms) of n-reach for processing 1 million randomly generated queries in synthetic datasets with $1K \leq |V| \leq 200$ K

$|V| \geq 10$ K. To explain this, we also verified that the index size at $|V| = 1$ K is almost the same as the L2 cache size, and at $|V| = 5$ K is slightly smaller than the L3 cache size, but the index size becomes larger than the cache size when $|V| \geq 10$ K. Thus, this result reveals that for handling very small datasets when their index size is smaller than the cache size, the index is likely kept in cache memory and hence it can give a remarkably short query time as small as 5 ns per query. However, we emphasize that the existing indexing methods that we compare with were all compiled using the same gcc compiler in the same way as our codes. Moreover, Table 21 shows that the index size of PWAH is smaller than that of n-reach in most cases, but Table 22 shows that n-reach still obtains shorter query time than PWAH in most cases.

6.10 Summary of experimental results

To summarize, we have the following main findings:

– Compared with the state-of-the-art indexes for processing shortest path distance queries, HCL [25] and PLL [3], k-reach is significantly more efficient and scalable for processing both k-hop reachability queries (as shown in Sect. 6.4) and shortest path distance queries (as shown in Sect. 6.8).
– The k-reach index is efficient, in terms of both indexing construction and query processing, for different values of k (as shown in Sect. 6.6).
– The k-reach index can be easily extended to process k-hop reachability queries of any k or even shortest path distance queries, with similar or only slightly degraded performance (as shown in Sect. 6.8).
– The k-reach index achieves reasonably good performance even for processing classic reachability queries, compared with the state-of-the-art indexes for processing classic reachability queries such as PTree [26], 3-hop [27], GRAIL [37], and PWAH [33].

7 Related work

A large number of indexes have been proposed for processing graph reachability queries [2,5–8,13–16,22,26–28,30,32–34,37,38]. We have analyzed these indexes and discussed why they are not suitable for processing k-hop reachability queries in Sect. 3. We have also discussed why the existing indexes for processing shortest path queries [3,12,16,25,35,36] are not efficient for processing k-hop reachability queries in Sect. 3.5.

Some other variations of graph reachability have also been proposed. For example, Jin et al. [24] studied distance-constraint reachability in uncertain graphs where the existence of an edge is given by a probability, and a query asks

Algorithm 6: Querying shortest path distance using scalable k-reach

```
Input : A more scalable $k$-reach index of $G$, consisting of $D_1 = (V_{D_1}, E_{D_1}, \omega_{D_1})$, $D_2 = (V_{D_2}, E_{D_2}, \omega_{D_2})$, a residual graph $D_3$, and two
        query vertices, $s$ and $t$
Output: A boolean indicator whether $s \rightarrow_k t$, and $d(s,t)$ if $s \rightarrow_k t$ is true
   //case 1: both $s$ and $t$ are in $S$
1  if $s \in S$ and $t \in S$ then
2      if $(s,t) \in E_{D_1}$ then
3          return true, and $\omega_{D_1}((s,t))$;
4      else
5          return false;
   //case 2: $s$ or $t$ is in $S$, but not both
6  else if $s \in S$ or $t \in S$ then
7      if $(s,t) \in E_{D_1}$ then
8          return true, and $\omega_{D_1}((s,t))$;
9      else if $\exists v \in S$ such that $\omega_{D_1}((s,v)) + \omega_{D_1}((v,t)) \leq k$ then
10         return true, and $\min\{\omega_{D_1}((s,v)) + \omega_{D_1}((v,t)) : v \in S\}$;
11     else
12         return false;
   //case 3: both $s$ and $t$ are in $S'$
13 else if $s \in S'$ and $t \in S'$ then
14     if $(s,t) \in E_{D_2}$ then
15         return true, and $\omega_{D_2}((s,t))$;
16     else if $\exists u, v \in S$ such that $\omega_{D_1}((s,u)) + \omega_{D_1}((u,v)) + \omega_{D_1}((v,t)) \leq k$ then
17         return true, and $\min\{\omega_{D_1}((s,u)) + \omega_{D_1}((u,v)) + \omega_{D_1}((v,t)) : u, v \in S\}$;
18     else
19         return false;
   //case 4: $s, t \notin S$ and $s$ or $t$ is in $S'$
20 else if $s \in S'$ or $t \in S'$ then
21     $d \leftarrow \infty$;
22     if $(s,t) \in E_{D_2}$ then
23         $d \leftarrow \min\{d, \omega_{D_2}((s,t))\}$;
24     if $\exists u, v \in S$ such that $\omega_{D_1}((s,u)) + \omega_{D_1}((u,v)) + \omega_{D_1}((v,t)) \leq k$, then
25         $d \leftarrow \min\{d, \min\{\omega_{D_1}((s,u)) + \omega_{D_1}((u,v)) + \omega_{D_1}((v,t)) : u, v \in S\}\}$;
26     if $\exists v \in S'$ such that $\omega_{D_2}((s,v)) + \omega_{D_2}((v,t)) \leq k$ then
27         $d \leftarrow \min\{d, \min\{\omega_{D_2}((s,v)) + \omega_{D_2}((v,t)) : v \in S\}\}$;
28     if $d \leq k$ then
29         return true, and $d$;
30     else
31         return false;
   //case 5: both $s$ and $t$ are not $S$ or $S'$
32 else if $s, t \in V_{D_3}$ then
33     $d \leftarrow \infty$;
34     if $\exists u, v \in S$ such that $\omega_{D_1}((s,u)) + \omega_{D_1}((u,v)) + \omega_{D_1}((v,t)) \leq k$, then
35         $d \leftarrow \min\{d, \min\{\omega_{D_1}((s,u)) + \omega_{D_1}((u,v)) + \omega_{D_1}((v,t)) : u, v \in S\}\}$;
36     if $\exists u, v \in S'$ such that $\omega_{D_2}((s,u)) + \omega_{D_2}((u,v)) + \omega_{D_2}((v,t)) \leq k$, then
37         $d \leftarrow \min\{d, \min\{\omega_{D_2}((s,u)) + \omega_{D_2}((u,v)) + \omega_{D_2}((v,t)) : u, v \in S'\}\}$;
38     if $d \leq k$ then
39         return true, and $d$;
40     else
41         start BFS from $s$ in $D_3$ and from $t$ in the reverse graph of $D_3$ in parallel for $\lceil k/2 \rceil$ hops: if the two BFSs meet at a common vertex $v$
           and $d(s,v) + d(t,v) \leq k$, return true, and $d(s,v) + d(t,v)$; otherwise, return false;
```

the probability that the distance from s to t is less than or equal to a user-defined threshold d in an uncertain graph. Their work focuses on designing probabilistic estimators for estimating the probability of reachability. Jin et al. also proposed constrained graph reachability by requiring edges on the path to have certain labels [23].

We are also aware of a recent work that applies the concept of vertex cover to construct an index for answering single-source shortest path distance queries [9]. They identified the limitation of vertex cover for processing shortest path distance queries and proposed a tree-structured index in which every node is a graph that keeps distance information. The k-reach graph has a similar limitation, i.e., it is a complete graph, if it is used for processing shortest path distance queries. However, the k-reach graph for k-hop reachability is a significantly smaller sparse graph. Moreover, their index is also too expensive for processing k-hop reachability queries.

Compared with the preliminary version of this paper [11], we proposed a more scalable method for handling large graphs. The preliminary version proposed to further reduce the size of the basic k-reach index by extending the coverage of the vertex cover to edges within h hops of a covering vertex. However, covering edges is a more rigid definition than covering vertices, which can be easily seen by comparing the size of a minimum vertex cover and that of a minimum dominating set (the latter is in general significantly smaller). Thus, for many real-world graphs, the size of the h-hop vertex cover may still be large. In this paper, we propose to employ a partial coverage as well as relax the 1-hop edge coverage in the classic vertex cover to k-hop vertex coverage, thus significantly reducing the index size for handling large graphs.

8 Conclusions

We proposed an efficient index, ***k*-reach**, to process k-hop reachability queries. The k-reach index is simple in design and easy to implement. In particular, the k-reach index can effectively handle skewed degree distribution in real-world graphs and is able to process both classic reachability queries (i.e., the case when $k = \infty$) and k-hop reachability queries. We analyzed the limitations of the existing works in handling k-hop reachability (see Sect. 3). Our experimental results verified the efficiency of k-reach in answering k-hop reachability queries, for both small and large values of k, thus demonstrating its suitability for different real-life applications where the value of k may vary. The results also showed that k-reach is significantly faster and more scalable than the state-of-the-art shortest path distance indexes [3,25]. In addition, we showed that k-reach is also efficient for processing classic reachability queries, as compared with the state-of-the-art indexes that are tailored for classic reachability [26,27,33,37].

Acknowledgments The authors would like to thank the reviewers for their constructive comments. This research is supported in part by CUHK direct grant 4055017, and grants from the Research Grants Council of the Hong Kong SAR (CUHK 411211 and CUHK 418512).

9 Appendix

Algorithm 6 presents an extension of Algorithm 4 to return the exact shortest path distance from s to t if s can reach t in k hops. The main difference is that in Algorithm 4, we can terminate the process as soon as we find that s can reach t in k hops in D_1 or D_2 or D_3, while in Algorithm 6 we need to continue with the intersection and join in order to find the minimum distance. However, the complexity of Algorithm 6 is the same as that of Algorithm 4 given in Sect. 5.5, since we give the worst case complexity in Sect. 5.5.

References

1. Abraham, I., Fiat, A., Goldberg, A.V., Werneck, R.F.F.: Highway dimension, shortest paths, and provably efficient algorithms. In SODA, pp. 782–793 (2010)
2. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. In SIGMOD Conference, pp. 253–262 (1989)
3. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. To appear in SIGMOD Conference (2013)
4. Boldi, P., Rosa, M., Vigna, S.: Hyperanf: approximating the neighbourhood function of very large graphs on a budget. In WWW, pp. 625–634 (2011)
5. Bramandia, R., Choi, B., Ng, W.K.: On incremental maintenance of 2-hop labeling of graphs. In WWW, pp. 845–854 (2008)
6. Chen, L., Gupta, A., Kurul, M.E.: Stack-based algorithms for pattern matching on DAGs. In VLDB, pp. 493–504 (2005)
7. Chen, Y., Chen, Y.: An efficient algorithm for answering graph reachability queries. In ICDE, pp. 893–902 (2008)
8. Chen, Y., Chen, Y.: Decomposing DAGs into spanning trees: A new way to compress transitive closures. In ICDE, pp. 1007–1018 (2011)
9. Cheng, J., Ke, Y., Chu, S., Cheng, C.: Efficient processing of distance queries in large graphs: A vertex cover approach. In SIGMOD Conference, pp. 457–468 (2012)
10. Cheng, J., Ke, Y., Fu, A.W.-C., Yu, J.X., Zhu, L.: Finding maximal cliques in massive networks by h*-graph. In SIGMOD Conference, pp. 447–458 (2010)
11. Cheng, J., Shang, Z., Cheng, H., Wang, H., Yu, J.X.: K-reach: Who is in your small world. PVLDB **5**(11), 1292–1303 (2012)
12. Cheng, J., Yu, J.X.: On-line exact shortest distance query processing. In EDBT, pp. 481–492 (2009)
13. Cheng, J., Yu, J.X., Lin, X., Wang, H., Yu, P.S.: Fast computation of reachability labeling for large graphs. In EDBT, pp. 961–979 (2006)
14. Cheng, J., Yu, J.X., Lin, X., Wang, H., Yu, P.S.: Fast computing reachability labelings for large graphs with high compression rate. In EDBT, pp. 193–204 (2008)
15. Cheng, J., Yu, J.X., Tang, N.: Fast reachability query processing. In DASFAA, pp. 674–688 (2006)
16. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. In SODA, pp. 937–946 (2002)

17. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press,Cambridge, MA (2009)
18. Dorogovtsev, S.N., Mendes, J.F.F., Samukhin, A.N.: Structure of growing networks with preferential linking. Phys. Rev. Lett. **85**(21), 4633–4636 (2000)
19. Faloutsos, M., Faloutsos, P., Faloutsos, C.: On power-law relationships of the internet topology. In SIGCOMM, pp. 251–262 (1999)
20. Feige, U.: A threshold of ln n for approximating set cover. J. ACM **45**(4), 634–652 (1998)
21. Hochbaum, D.S.: Approximation Algorithms for NP-hard Problems. PWS Publishing Co., Boston, MA, USA (1997)
22. Jagadish, H.V.: A compression technique to materialize transitive closure. ACM Trans. Database Syst. **15**(4), 558–598 (1990)
23. Jin, R., Hong, H., Wang, H., Ruan, N., Xiang, Y.: Computing label-constraint reachability in graph databases. In SIGMOD Conference, pp. 123–134 (2010)
24. Jin, R., Liu, L., Ding, B., Wang, H.: Distance-constraint reachability computation in uncertain graphs. PVLDB **4**(9), 551–562 (2011)
25. Jin, R., Ruan, N., Xiang, Y., Lee, V.E.: A highway-centric labeling approach for answering distance queries on large sparse graphs. In SIGMOD Conference, pp. 445–456 (2012)
26. Jin, R., Ruan, N., Xiang, Y., Wang, H.: Path-tree: An efficient reachability indexing scheme for large directed graphs. ACM Trans. Database Syst. **36**(1), 7 (2011)
27. Jin, R., Xiang, Y., Ruan, N., Fuhry, D.: 3-hop: a high-compression indexing scheme for reachability query. In SIGMOD Conference, pp. 813–826 (2009)
28. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficiently answering reachability queries on very large directed graphs. In SIGMOD Conference, pp. 595–608 (2008)
29. Newman, M.E.J.: The structure and function of complex networks. SIAM Rev. **45**(2), 167–256 (2003)
30. Simon, K.: An improved algorithm for transitive closure on acyclic digraphs. Theor. Comput. Sci. **58**(1–3), 325–346 (1988)
31. Stann, F., Heidemann, J.: Rmst: Reliable data transport in sensor networks. In 1st IEEE International Workshop on Sensor Net Protocols and Applications, pp. 102–112 (2003)
32. Trißl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In SIGMOD Conference, pp. 845–856 (2007)
33. van Schaik, S.J., de Moor, O.: A memory efficient reachability data structure through bit vector compression. In SIGMOD Conference, pp. 913–924 (2011)
34. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: Answering graph reachability queries in constant time. In ICDE, p. 75 (2006)
35. Wei, F.: TEDI: Efficient shortest path query answering on graphs. In SIGMOD Conference, pp. 99–110 (2010)
36. Xiao, Y., Wu, W., Pei, J., Wang, W., He, Z.: Efficiently indexing shortest paths by exploiting symmetry in graphs. In EDBT, pp. 493–504 (2009)
37. Yildirim, H., Chaoji, V., Zaki, M.J.: Grail: Scalable reachability index for large graphs. PVLDB **3**(1), 276–284 (2010)
38. Zhu, L., Choi, B., He, B., Yu, J.X., Ng, W.K.: A uniform framework for ad-hoc indexes to answer reachability queries on large graphs. In DASFAA, pp. 138–152 (2009)