A Uniform Framework for Ad-Hoc Indexes to Answer Reachability Queries on Large Graphs

Linhong Zhu[1], Byron Choi[2], Bingsheng He[3], Jeffrey Xu Yu[4], and Wee Keong Ng[1]

[1] Nanyang Technological University, Singapore
{ZHUL0003,AWKNG}@ntu.edu.sg
[2] Hong Kong Baptist University, China
choi@hkbu.edu.hk
[3] Microsoft Research Asia
savenhe@microsoft.com
[4] Chinese University of Hong Kong, China
yu@se.cuhk.edu.hk

Abstract. Graph-structured databases and related problems such as reachability query processing have been increasingly relevant to many applications such as XML databases, biological databases, social network analysis and the Semantic Web. To efficiently evaluate reachability queries on large graph-structured databases, there has been a host of recent research on graph indexing. To date, reachability indexes are generally applied to the entire graph. This can often be suboptimal if the graph is large or/and its subgraphs are diverse in structure. In this paper, we propose a uniform framework to support existing reachability indexing for subgraphs of a given graph. This in turn supports fast reachability query processing in large graph-structured databases. The contributions of our uniform framework are as follows: (1) We formally define a graph framework that facilitates indexing subgraphs, as opposed to the entire graph. (2) We propose a heuristic algorithm to partition a given graph into subgraphs for indexing. (3) We demonstrate how reachability queries are evaluated in the graph framework. Our preliminary experimental results showed that the framework yields a smaller total index size and is more efficient in processing reachability queries on large graphs than a fixed index scheme on the entire graphs.

1 Introduction

Recent interests on XML, biological databases, social network analysis, the Semantic Web, Web ontology and many other emerging applications have sparked renewed interests on graph-structured databases (or simply *graphs*) and related problems (e.g., query processing and optimization). In this paper, we focus on querying large graphs. In particular, we are interested in a kind of fundamental queries from classical graph-structured databases – reachability query. Specifically, given two vertices u and v, a reachability query returns true if and only if there is a directed path from u and v (denoted $u \rightsquigarrow v$); otherwise, the query returns false.

Reachability queries have many emerging applications in graph-structured databases. For example, in XML, the ancestor and descendant axes of XPATH can be implemented with reachability queries on the graph representation of XML. Reachability queries are

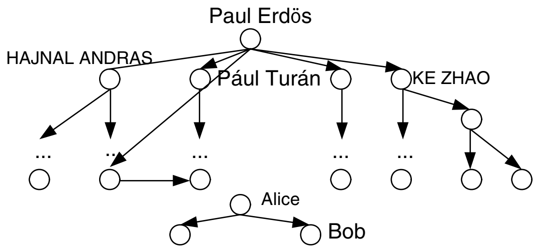


Fig. 1. An example of a real graph

also useful for building a query language [1] for the Semantic Web. As required by these applications, it is desirable to have efficient reachability query processing.

Example 1. Consider a directed graph that represents a social network of a set of researchers, where the vertices correspond to researchers and the edges correspond to the co-authorship relationship between two researchers, as shown in Figure 1. Social network analysis may often require reachability queries. For example, we may ask whether a researcher "Alice" has an *Erdös number*. A simple way to answer this query is to check whether "Alice" is reachable from "Paul Erdös".

To provide some background on reachability queries, we review existing naïve evaluation algorithms for reachability queries and the indexes for different kinds of graphs. There are two naïve alternatives for evaluating reachability queries on a graph: (1) A reachability query can be evaluated using a traversal of the graph. The runtime is $O(|G|)$, where $|G|$ denotes the size of the graph G. (2) A reachability query can also be evaluated by precomputing the transitive closure of the graph, whose size is quadratic to the graph size in the worst case. A reachability query can then be a simple selection on the transitive closure. It is clear that these two approaches are not scalable. Much indexing technique has been proposed for optimizing reachability queries on trees [2,3], directed acyclic graphs (DAGs) [4,5,6,7,8], and arbitrary graphs [9,10,11] (see Section 5). These indexes have demonstrated some performance improvement on the graphs with certain structural characteristics.

Unlike relational data, graph-structured data may vary greatly in its structure; e.g., trees, sparse/dense DAGs and sparse/dense cyclic graphs. It is evident that the structure of the graphs has an impact on the performance of reachability indexes on graphs. For instance, dual labeling [11] works best for sparse graphs but performs suboptimally on dense graphs. Hence, a single reachability index is sometimes not ideal to graphs that have different structures. Given these, we raise the following issues and propose some solutions for these issues in this paper:

1. A notion of data granularity is missing in graph-structured databases. Let us consider an example from relational databases. One may build a B+ tree on a *subset* of the attributes of a relation for range queries and a hash index on some other subsets of attributes for equi-joins. In comparison, to date, a graph index (such as dual labeling [11]) is either applied to the entire graph, or not applied at all. Is there a

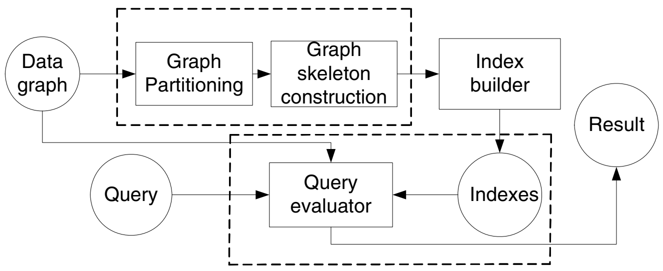


Fig. 2. Overview of our uniform framework for querying large graphs

general approach to seamlessly support multiple indexes? For example, one may apply dual labeling [11] to one subgraph and 2-hop [10] to another?

2. Structural differences between subgraphs have not been considered by state-of-the-art reachability indexes. Many real graphs such as Web graphs, telephone call graphs, and global social network graphs have different structures, both locally and globally. Real graphs, as a whole, are typically sparse with a constant average degree [12]. However, there may be local dense subgraphs. Hence, we need to address structural differences and determine suitable indexing techniques for subgraphs. Is it possible to detect different substructures from a graph and apply suitable indexes to these substructures?
3. Different reachability indexing techniques require different query evaluation algorithms. Is it possible to support multiple indexes and yet reuse existing query evaluation algorithms without modifying the indexes?

To address the above issues, we propose a uniform framework for indexing graphs. With the framework, we can flexibly use any existing index for reachability queries on subgraphs of a graph. An overview of our framework is shown in Figure 2. Our framework consists of two components: (1) Graph framework construction through graph partitioning and (2) reachability query evaluation on the graph framework with different indexes. As a proof of concept, our current prototype supports two state-of-the-art indexes for reachability queries, namely Interval [7] and HOPI [13].

In summary, we define a graph framework to represent a graph as a set of partitions and a graph skeleton. Each partition can use any existing reachability index. In conjunction with the framework, we define a cost function and propose a heuristic algorithm for graph partitioning. We illustrate how existing query evaluation techniques can be extended to our graph framework. In particular, a reachability query is casted into inter-partition and intra-partition reachability queries on indexes. We present our experimental evaluation on our framework with both synthetic and real graphs.

The remainder of the paper is organized as follows: In Section 2, we define notations used in this paper, our graph framework and the evaluation of reachability queries using the graph framework. The graph framework construction is presented in Section 3. In Section 4, we present an experimental study to evaluate the effectiveness and efficiency of our approach. Related work is discussed in Section 5. We conclude this work and present future work in Section 6.

2 Labeling Graph Framework for Reachability Query

In this section, we define our *graph framework* to represent reachability information of an arbitrary graph. In brief, the graph framework comprises information of strongly connected components, partitions of a graph and connections between partitions. We evaluate reachability queries efficiently by applying multiple indexing techniques to the graph framework. We defer the details of the construction of the graph framework to Section 3.

We describe the notations used in this work in Section 2.1. In Section 2.2, we present the definition of our graph framework and its properties. In Section 2.3, we show how to apply multiple indexes to the graph framework and how to process reachability queries in the graph framework.

2.1 Preliminaries

We denote a directed graph to be $G = (V, E)$, where V is a (finite) set of vertices, and E is a (finite) set of directed edges representing the connection between two vertices. We define an auxiliary function $\texttt{reach}(v_1, v_2)$ that returns true *iff* v_1 can reach v_2.

Definition 1. *The* condensed graph *of G is denoted as $G^* = (V^*, E^*)$, where a vertex $v_i^* \in V^*$ represents a strongly connected component C_i in G and each edge $(v_i, v_j) \in E^*$* iff *there is at least one edge $(u, v) \in E$ such that $u \in C_i$ and $v \in C_j$.*

The condensed graph can be computed efficiently using Tarjan's algorithm with time complexity $O(|V|+|E|)$ [19].

We use G_i to denote a subgraph of G and V_i to denote the set of vertices in G_i. We define a (non-overlapping) partitioning of graph as follows:

Definition 2. *A partitioning of graph G P(G) is $\{G_1, G_2, ..., G_k\}$, where $\forall\, i \in$ [1...k], $k \leq |V|$, $\cup_{i=1}^{k} V_i = V$, $V_i \cap V_j = \emptyset$, where $i \neq j$.*

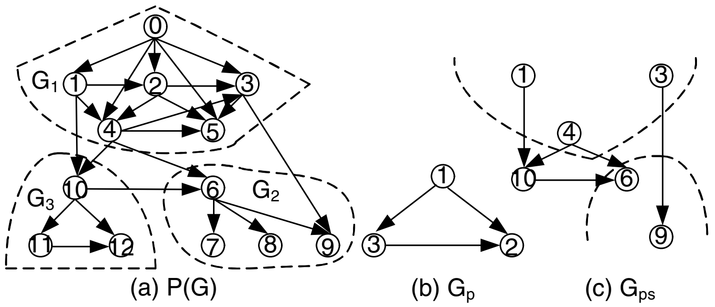
Example 2. Consider the graph shown in Figure 3(a). We partition the graph on the left into three partitions V_1=$\{0, 1, 2, 3, 4, 5\}$, V_2=$\{6, 7, 8, 9\}$ and V_3=$\{10, 11, 12\}$. G_1, G_2 and G_3 is a dense subgraph, a subtree and a sparse subgraph, respectively.

Based on this partitioning, we define a *partition-level graph* as follows.

Definition 3. *Given a partitioning P(G), the* partition-level graph *$G_p(G)$ is (V_p, E_p), where each vertex $v_i \in V_p$ represents a partition G_i in P(G) and an edge (v_i, v_j) $\in E_p$* if *there is an edge (u, v) $\in E$ such that u and v are vertices in Partitions G_i and G_j, respectively.*

Example 3. Consider the graph G and its partitioning $P(G)$ in Example 2. The partition-level graph $G_p(G)$ is shown in Figure 3(b). Vertices 1, 2 and 3 in G_p represent Partitions G_1, G_2 and G_3, respectively.

Next, let us consider the relationships between two partitions G_i and G_j that has not been captured by the partition-level graph. We define a *partition-level skeleton graph* $G_{ps}(G)$ to capture the connecting vertices of partitions of a graph G.



pid	$\mathcal{M}$ (pid)
1	$\{1, 3, 4\}$
2	$\{6, 9\}$
3	$\{10\}$

Fig. 3. A partitioning of a graph and its graph skeleton

Definition 4. *Given a partition-level graph $G_p(G)$, the* partition-level skeleton graph *$G_{ps}(G)$ is (V_{ps}, E_{ps}), where $V_{ps} \subseteq V$, $v \in V_{ps}$ is either a source or a target vertex of an inter-partition edge and E_{ps} is the set of inter-partition edges.*

With the above, a graph skeleton is defined as follows:

Definition 5. *Given a graph G and a partitioning of G, $P(G)$, the* graph skeleton *$S(G)$ is a 3-ary tuple (G_p, G_{ps}, $\mathcal{M}$), where G_p=(V_p, E_p) is the partition-level graph, G_{ps}=(V_{ps}, E_{ps}) is the partition-skeleton graph and $\mathcal{M} : V_p \rightarrow 2^{V_{ps}}$ is a mapping function that takes a partition v_i as input and gives a subset of V_{ps}: $\{v \mid v \in V_{ps}$ and $v \in G_i\}$ as output.*

To illustrate the mapping function $\mathcal{M}$, we give an example in Figure 3.

2.2 Graph Framework

Given the previous preliminary definitions, we propose our graph framework.

Definition 6. *A* graph framework *of a graph G, denoted as $H(G)$, consists of a strongly connected component index $C(V)$, a partitioning of the condensed graph $P(G^*)$ and a graph skeleton $S(G^*)$: $H(G)$= ($C(V)$,$P(G^*)$, $S(G^*)$).*

We remark that the proposed graph framework has the following properties:

– The graph framework supports multiple reachability indexing techniques. Different partitions may use different indexing techniques.
– The graph skeleton, which contains the inter-partition information, can be indexed in order to facilitate query processing. For example, we could apply the hash index to accelerate the search on the mapping $\mathcal{M}$ and any reachability indexes to the graph G_p and G_{ps} in $S(G^*)$.
– The graph framework consists of indexed subgraphs. Hence, the query evaluation on the graph framework can be transformed into the query evaluation on relevant indexed subgraphs.

2.3 Query Processing on Graph Framework with Reachability Indexing

In this subsection, we present an algorithm for evaluating a reachability query on our graph framework with multiple reachability indexes, as shown in Algorithm 1.

Algorithm 1. A query evaluation algorithm on the graph framework `evaluate-query`

```
Input: a graph framework $H(G)$: (C(V), $P(G^*)$, $S(G^*)$) with indexes, two input vertices $x$ and $y$,
        where $x, y \in V$
Output: true if $x$ can reach $y$, false otherwise
1: if C(x)==C(y)
2:     return true
3: denote $G_i(G_j)$ to be the partition of $x(y)$
4: if $G_i = G_j$
5:     return true if $x \rightsquigarrow y$ in $G_i$, false otherwise
6: if $G_i \not\rightsquigarrow G_j$ with respect to $G_p$ in $S(G^*)$
7:     return false
8: else $V_i$=$\mathcal{M}(i)$, $V_j$=$\mathcal{M}(j)$
9:     for each vertex $v_i \in V_i$
10:        for each vertex $v_j \in V_j$
11:            if $v_i \rightsquigarrow v_j$ with respect to $G_s$ in $S(G^*)$
12:                return true if $x \rightsquigarrow v_i$ in $G_i$ and $v_j \rightsquigarrow y$ in $G_j$
13: return false
```

Algorithm `evaluate-query` returns true if vertex x is able to reach vertex y. It returns false otherwise. The first step is to obtain the strongly connected component for vertices x and y. If they are in the same strongly connected component, the query returns true (Lines 1-2). Next, we compute the partitions where x and y reside, i.e., G_i and G_j (Line 3). If the two input vertices are in the same partition, we use the index of the partition to answer the reachability query (Lines 4-5). Next, if we find that G_i cannot reach G_j by using the index of G_p in $S(G^*)$, then u is not able to reach v and the query returns false (Lines 6-7). Otherwise, we apply the mapping $\mathcal{M}$ to obtain the set of vertices in G_s related to partitions G_i and G_j, i.e., V_i and V_j (Line 8). We test whether there is a vertex $v_i \in V_i$ that is able to reach V_j. If so, we return true if x is able to reach v_i in G_i and v_j reaches y in G_j (Lines 9-12). Otherwise, we return false (Line 13). The correctness of this algorithm can be easily derived from the definition of the graph framework.

Complexity. The time complexity of query processing on graph framework is index-dependant, i.e., it is determined by the reachability indexes applied to the graph framework. For example, assume that vertices x and y are not in the same partition and partition i, containing x, is indexed with interval labeling [7] with query time complexity $O(|V_i|)$ and partition j, containing y, is indexed with HOPI labeling [20] with query time complexity $O(|E_j|^{1/2})$. The partition-level graph G_p and partition-level graph G_s are indexed with dual labeling [11] with constant query time complexity. Hence, in this particular example, the overall complexity of `evaluate-query` is

$O(|E_{ij}|(|V_i|+|E_j|^{1/2}))$, where E_{ij} denotes the set of inter-partition edges connecting the partition i and partition j.

3 Graph Framework Construction

Given an input graph G, we construct a graph framework as follows: (1) compute condensed graph G^*; (2) compute a partitioning of G^*, $P(G^*)$, using a heuristic algorithm proposed in Section 3.1; (3) based on $P(G^*)$, compute the graph skeleton $S(G^*)$. The key in graph framework construction is to compute $P(G^*)$. Once $P(G)$ is computed, the graph skeleton $S(G)$ can simply be constructed as follows:

1. compute the partition-level graph G_p;
2. compute the set of inter-partition edges E_I;
3. compute the subgraph G_s which is induced by the edge set E_I; and
4. compute the mapping $\mathcal{M}$ between a partition and a set of vertices in G_s.

Hence, in this section, we focus on the details of determining $P(G^*)$ of a given graph G. First, we propose the objective function of our graph partitioning problem and the heuristic algorithmic strategy to solve the problem. Next, we present how to estimate the query cost of a graph and a graph framework, which is used in the heuristic graph partitioning.

3.1 Heuristics for Graph Partitioning

In this subsection, we present the objectivity of our graph partitioning problem. Next, we present the overall procedure for constructing the graph framework.

The graph partitioning problem considered in this work can be formulated as follows: Given a graph G=(V, E), determine k non-overlapping subsets of V_1,..., V_k such that:

1. $\cup_{i=1}^{k} V_i = V$, $V_i \cap V_j = \emptyset$ where $i \neq j$; and
2. the estimation of query time; i.e., the number of labels accessed during query processing, is minimal,

where k is determined during partitioning.

Overall algorithm. Let $E(G)$ and $E(H(G))$ denote the query costs on a graph G and a graph framework $H(G)$, respectively. Our heuristic graph partition algorithm works as follows: Given a k-partitioning of a graph $P(G)$ and its corresponding graph framework $H(G)$, we create a new $(k{+}1)^{\mathrm{th}}$ partition for a set of vertices $P'(G)$ if the resulting new graph framework $H'(G)$ reduces the query cost; i.e., $E(H'(G)) \leq E(H(G))$ (shown in Algorithm 2). Hence, in Algorithm 2, the main strategy is to determine whether further partitioning reduces the cost.

More specifically, we start with a partitioning of G that all the vertices are in the same partition (Line 1). Consider the current k-partition $P(G)$. We find the partition G_{max} whose cost is maximum among all other partitions in $P(G)$ (Line 3). Next, we assign a set of vertices in G' to a new partition G_{k+1}: for every vertex v in G_{max}, we

Algorithm 2. The heuristic graph partition algorithm `partition`(G)

```
Input: a digraph $G$
Output: a partitioning of $G$, $P(G)$
1: let $k$=1, $C_{\texttt{pre}}$=$E(G)$, $C_{\texttt{decre}}$=0, $P(G)$=$\{G\}$
2: do
3:     $G_{max}$=$\texttt{arg}\max_{1\leq i\leq k}\{E(G_i)\}$, $G_i \in P(G)$
4:     $V_{k+1} = \emptyset$
5:     for each vertex $v$ of $G_{max}$
6:         if $E(G_{max} \setminus v) < E(G_{max})$
7:             $V_{k+1} = V_{k+1} \cup v$
8:     $k+1$-partitioning $P'(G)$=$\{P(G) \setminus V_{k+1}, G_{k+1}\}$
           /* where $G_{k+1}$ is the subgraph induced by $V_{k+1}$ */
9:     call refinement procedure, $P(G)$=Refinement($G$, $P'(G)$)
10:    $C_{\texttt{decre}}$=$E(P(G))$-$C_{\texttt{pre}}$, $C_{\texttt{pre}}$=$E(P(G))$, $k = k+1$
11: while $C_{\texttt{decre}}$ <0
12: return $P(G)$
```

place it into the new partition *if* its removal from G_{max} decreases its cost (Lines 5-7). This results in a $k+1$-partitioning $P'(G)$ (Line 8). In order to optimize the quality of the $k+1$ partitions, we invoke the partition refinement procedure `Refinement`(G, $P'(G)$) (Line 9) to obtain $P_r(G)$. We proceed to the next partitioning iteration if the cost has been reduced in the current iteration. Otherwise, Algorithm 2 terminates and returns $P_r(G)$ (Lines 10-12).

Partition refinement. Next, we present the details of the refinement procedure (Algorithm 3) used in Algorithm 2. In the refinement procedure, we improve the quality of a given k-partitioning. We apply a search technique to find a better k-partitioning with a lower cost. Initially, Algorithm 3 starts with a partitioning of graph $P(G)$ (Line 1). Then, for each vertex, we search for a good assignment that minimizes the cost: We find the best target partition `pid` for a vertex v such that the new k-partitioning produced by assigning v to Partition `pid` has the minimal cost among all other k-partitionings produced by other assignments (Lines 3-6). The partitioning algorithm terminates if the new partitioning found does not decrease the cost or the iteration number is up to a user input value m; otherwise, Algorithm 2 search for other possible assignments (Lines 7-9).

Complexity. Let the number of iterations needed in the refinement procedure be m. The complexity of the whole partition procedure is $O(mk^2(|V| + |E|))$, where k is the number of partitions. We remark that the values of m and k are often small in real applications.

3.2 Query Cost Estimation

In this subsection, we discuss how to model the costs, $E(G)$ and $E(H(G))$. One of the important properties of our framework is that it is able to support multiple reachability indexes, where any single specified reachability index is a special case of our

Algorithm 3. Partition Refinement `Refinement`$(G, P(G))$

```
Input: a digraph $G$, initial $k$-partitioning $P(G)$, iteration number $m$
Output: a new $k$-partitioning of $G$, $P_r(G)$
1: $C_{\texttt{pre}}$=$E(P(G))$, $C_{\texttt{decre}}$=0, $i$=0
2: do
3:     for each vertex $v \in V$ of $G$
4:         let $P_j(G)$ denotes a new partitioning resulted by removing $v$ into partition $j$ in $P(G)$
5:         pid($v$)=arg $\min_{1 \le j \le k} \{E(P_j(G)) - E(P(G))\}$
6:         $P(G)$=$P_{\texttt{pid}}(G)$
7:     $C_{\texttt{decre}}$=E($P(G)$)-$C_{\texttt{pre}}$, $C_{\texttt{pre}}$=E($P(G)$), $i$++
8: while $C_{\texttt{decre}}$ <0 and $i < m$
9: return $P(G)$
```

framework. However, the accuracy of $E(G)$ or $E(H(G))$ is highly dependant on the cost model of reachability indexes involved. Hence, to compute the value of $E(G)$ and $E(H(G))$, the pre-condition is that involved reachability indexes have a reasonable cost model. As an illustration, we implement two state-of-the-art indexes, Interval [7] and HOPI [13] in our prototype of the graph framework. It has been known that the time complexity for a reachability query on the HOPI and Interval indexes are $O(|E|^{\frac{1}{2}})$ and $O(|V|)$, respectively. Therefore, the query time estimation for $E(G)$ can be modeled by Equation 1.

$$E(G) = \min(C_1|V|, C_2|E|^{\frac{1}{2}}), \tag{1}$$

where C_1 and C_2 are the unit cost in real measurements.

Based on $E(G)$, the estimated query time on $H(G)$, which consists of a k-partitioning $P(G)$ and a graph skeleton $S(G)$, is defined as follows:

$$E(H(G)) = \sum_{i=1}^{k} E(G_i) + E(G_p) + E(G_s), \tag{2}$$

where k is the number of partitions.

4 Experimental Evaluation

In this section, we perform an experimental study to evaluate the effectiveness and efficiency of our proposed techniques. All experiments were run on a machine with a 3.4GHZ CPU. The run-times reported are the CPU times. We implemented all the proposed techniques in C++. Regarding graph indexes, we used the HOPI implementation from [13] and we implemented the Interval scheme for DAGs [7]. We also used the implementation of a recent path-tree approach from Jin *et al.* [8].

4.1 Index Size and Query Performance Evaluation on Real Data

We used a collection of real graphs in this experiment. We report the statistics of the real graphs in Table 1. Among them, “days”, “hep-th-new” and “eatRS” are obtained from a Web graph repository [21]; and the other real graphs are provided by Jin *et al.* [8].

Table 1. Statistics of real graphs

Tree-like graphs	$\|V\|$	$\|E\|$	$\|V\|^*$	$\|E\|^*$	Other graphs	$\|V\|$	$\|E\|$	$\|V\|^*$	$\|E\|^*$
kegg	14271	35170	3617	3908	days	13332	243447	13332	148038
vchocyc	10694	14207	9491	10143	hep-th-new	27770	352807	20086	130469
mtbrv	10697	13922	9602	10245	eatRS	23219	325624	15466	19916
agrocyc	13969	17694	12684	13408	hpycyc	5565	8474	4771	5859
anthra	13736	17307	12499	13104	nasa	5704	7942	5605	6537
human	40051	43879	38811	39576	xmark	6483	7654	6080	7025

Table 2. Index size comparison on real graphs.

Tree-like graphs	HOPI	Interval	Ptree	Our	Other graphs	HOPI	Interval	Ptree	Our
kegg	9488	10078	1703	2884	days	199788	227364	–	199826
vchocyc	33920	20196	830	1216	hep-th-new	268524	244748	–	204802
mtbrv	34312	20406	812	1204	eatRS	31032	67952	–	31034
agrocyc	43664	26728	962	1362	hpycyc	16576	11658	4224	**2118**
anthra	42888	26146	733	1160	nasa	49954	12852	5063	**1644**
human	84916	79058	965	1438	xmark	44112	14038	2356	**1880**

In the first set of experiments, we compared the index size of our graph framework with two popular approaches – Interval [7] and HOPI [20] and a recent path-tree approach [8] (PTree). The results are shown in Table 2. The reported index size is the number of integers in the indexes.

From the results presented in Table 2, we found that the index size of our approach is clearly smaller than that of the HOPI and Interval approaches. The reason is simple: for each subgraph in our framework, our approach chose a relatively better one between the two approaches. In addition, our approach is comparable to the Ptree approach. For graphs that are not "tree-like", our approach achieved a much smaller index size than the Ptree approach (the bold numbers in Table 2). Although our approach is sometimes worse than the Ptree approach for "tree-like" graphs, our approach is more general than the Ptree approach since the Ptree approach could be a special case of our approach (where the Ptree approach is applied to all partitions). However, since an accurate cost model for the Ptree approach has not been available, we did not apply the Ptree approach to any partition (subgraph) in our framework.

Next, we investigate the time for index construction and query processing. Here we only compared our methods with HOPI and the Interval approach. This is because the CPU time measurer and the query evaluator of these three approaches are all implemented by us while the Ptree approach is provided by its authors. All three approaches run on MS Windows while the Ptree approach runs on Linux. Hence, a comparison of indexing and query time between our approach and the Ptree approach may be affected by many implementation issues. When comparing the index construction time and the query time, we used two metrics, *i.e.*, S_L, and S_H to evaluate our approach, where $S_L = \frac{\min(\text{HOPI, Interval})}{\text{O}ur}$, and $S_H = \frac{\max(\text{HOPI, Interval})}{\text{O}ur}$. S_L measures the performance

Table 3. Construction time comparison on real graphs (ms)

Tree-like graphs	HOPI	Interval	Our	S_L	S_H	Other graphs	HOPI	Interval	Our	S_L	S_H
kegg	473.694	3.92742	4.19078	0.94	113	days	9484.38	93.113	9096.3502	0.01	1.04
vchocyc	1125	7.22463	7.3448	0.98	153	hep-th-new	56281.3	66.7182	71.7037	0.93	785
mtbrv	1140.63	5.47139	8.26619	0.66	138	eatRS	1718.75	33.1035	1728.2479	0.02	0.995
agrocyc	1500	7.18841	9.66451	0.74	155	hpycyc	609.375	4.52743	5.76455	0.79	106
anthra	1453.13	7.01311	9.72699	0.72	149	nasa	1203.13	3.46655	4.4054	0.79	273
human	4343.75	38.9534	59.8715	0.65	73	xmark	1000	3.80507	4.83549	0.79	207

Table 4. Total query time comparison on real graphs (ms)

Tree-like graphs	HOPI	Interval	Our	S_L	S_H	Other graphs	HOPI	Interval	Our	S_L	S_H
kegg	1315.07	1107.55	1097.85	1.008	1.2	days	7854.39	6645.64	7929.75	0.84	0.99
vchocyc	4136.98	2667.57	2674.55	0.997	1.5	hep-th-new	5576.43	7265.8	5641.19	0.99	1.3
mtbrv	4128.25	2672.27	2738.87	0.98	1.5	eatRS	3397.71	3514.07	3442.64	0.987	1.02
agrocyc	4286.65	2839.21	2797.93	1.015	1.5	hpycyc	3818.74	2599.91	2520.07	1.03	1.5
anthra	4254.57	2782.02	2767.95	1.005	1.5	nasa	5672.01	2739.5	2720.18	1.007	2.08
human	5003.2	3957.09	3901.01	1.014	1.28	xmark	5178.59	2677.69	2698.7	0.99	1.91

comparison between our method and the best of the two static methods. S_H measures the performance gain if an inefficient static method is chosen.

Table 3 illustrates that the Interval approach requires two traversals to construct the index and therefore always has the smallest indexing construction time. As known from previous work [9], the HOPI indexing time for large graphs can often be costly. Since a combination of HOPI and Interval is applied to our prototype implementation, the construction time of our approach is roughly between these two. In particular, the time depends on the percentage of the partitions using each of these indexing approaches. If most of the partitions are using HOPI, the construction time is closer to HOPI, such as the graph "days" and "eatRS", as shown in Table 3.

To study query performance, we issue one million random reachability queries on the indexes constructed for the real graphs. We used an in-memory IO simulation to estimate the IO cost. The IO simulation performs the following: Reachability labels (i.e., indexes) are stored in pages with the size 4KB. During query processing, we maintain a buffer with the size 4MB. When we check whether two vertices are reachable, we first obtain the ID of pages where the labels of two vertices are kept. Then we access the buffer to read the labels from required pages. If those pages are in buffer, we read the labels from pages directly. Otherwise, before reading labels from those pages, we insert each page into buffer or replace an old page in buffer using LRU replacement policy. Finally, we report the total query time and the number of labels accessed during query processing in Table 4 and 5, respectively. The average values of S_L in the total query time

Table 5. Number of labels accessed during query processing on real graphs

Tree-like graphs	HOPI	Interval	Our	S_L	S_H	Other graphs	HOPI	Interval	Our	S_L	S_H
kegg	663830	415144	407876	1.02	1.63	days	14986610	32996064	14957887	1.0	2.2
vchocyc	3251476	246004	242602	1.02	13.4	hep-th-new	8869499	47682220	8749328	1.01	5.45
mtbrv	3285414	248322	245036	1.01	13.4	eatRS	1333936	3181514	1333963	1.0	2.39
agrocyc	3292830	241414	237890	1.01	13.8	hpycyc	3017708	787830	770372	1.02	3.92
anthra	3298963	205832	206216	0.99	16	nasa	8993648	655730	655716	1.0	13.72
human	2160820	78618	80188	0.98	27	xmark	6838694	583146	583262	1.0	11.72

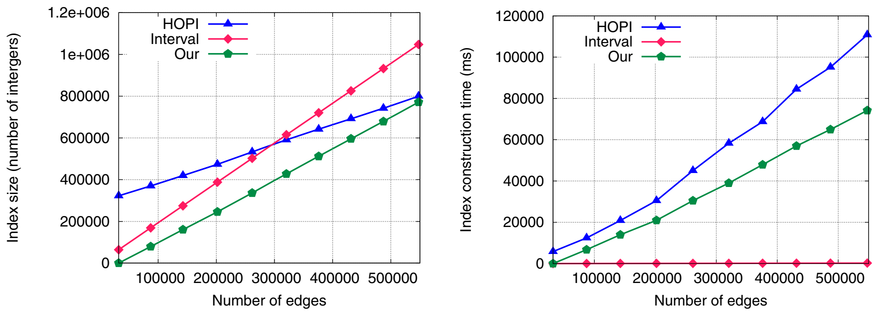


Fig. 4. Index size and construction time comparison on synthetic data

and the number of labels accessed are 0.99 and 1.006, respectively. These indicate that our approach is comparable to (or slightly better than) the best of Interval and HOPI approaches in the total query time and the number of labels accessed. Moreover, the average values of S_H in the total query time and the number of labels accessed are 1.44 and 10.38, respectively. This supports that our approach avoids the cost of selecting an inefficient static method. In all, our approach is both IO efficient and time efficient.

4.2 Index Size and Query Performance Evaluation on Synthetic Data

In Section 4.1, we compared our approach with Interval, HOPI and the Ptree approach on real graphs. In order to have a full control over graph structures, we implemented our own graph generator which controls the percentage of tree-like components and graph-like components. The generator works as follows: First, we generate a set of tree-like DAGs and a set of dense DAGs with the maximum fan-out of spanning tree F=6, the branch depth of spanning tree D=6. Then, we generate a large graph with n vertices by connecting p ($\times 100\%$) dense graphs to 1-p ($\times 100\%$) of tree-like graphs, where n and p are the two input parameters.

We generate a set of random graphs with n around 30k, and varying p from 0 to 0.9. The indexing size and index construction time comparison of Interval, HOPI, and our

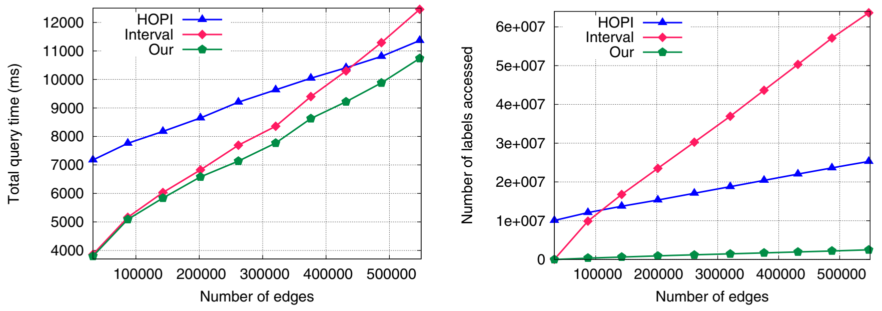


Fig. 5. Total query time and number of labels accessed comparison on synthetic data

approach is shown in Figure 4[1]. Similar to the previous experiment, the index size of our approach is significantly better than HOPI and Interval. Regarding the construction time, we find that the construction time of our approach is comparable to the Interval approach when the graph is sparse and slightly worse than the Interval approach when the graph is dense. However, the construction time of our approach is much smaller than that of the HOPI approach.

Figure 5 shows the total query time and the number of labels accessed of one million random queries. It is clear that our approach has a better query performance and is more IO efficient than the best of HOPI and Interval.

5 Related Work

There has been a large body of work on indexing for reachability queries on graphs. Due to space constraints, we list a few (non-exhaustive) examples in this section.

Dietz [3] assigns an interval to each vertex in a tree. A vertex can reach another vertex *iff* its interval is properly contained in the interval of the other vertex. There has been a host of work that demonstrates good query performance of the interval approach. Wu *et al.* [2] propose to use prime numbers to encode reachability information of a tree. A vertex is labeled with a product of prime numbers. A vertex is reachable from another vertex *iff* its label is divisible by the label of the other vertex. Wu and Zhang [5] extend this work [2] to support DAGs. Wang *et al.* [11] combine the interval approach for trees and a technique for indexing non-tree edges of a graph. The technique has a constant query time and small index size. Schenkel [10] and Cheng [13] extend *2-hop* labeling scheme, originally proposed by Cohen *et al.* [9], to efficiently index a large collection of XML documents. Trißl *et al.* [14] propose an efficient relational-based implementation that bases on the interval and *2-hop* labelings to index directed graphs.

All the aforementioned techniques index an entire graph. In contrast, our work focuses on a framework that supports applying different indexing techniques to different subgraphs. Hence, this work is orthogonal to any specific indexes.

[1] In this experiment, we did not compare the index size of our approach with Ptree [8], as our files storing the random graphs cannot be recognized by the Ptree implementation in Linux.

Perhaps [15,8] are the most relevant works. In [15], it proposes a hierarchical labeling scheme that identifies spanning trees for the interval labelings [3] and dense subgraphs in remainder graphs for 2-hop labeling [9]. While the problem being studied is similar to ours, their approach is tightly coupled with interval labelings and compression of the reachability matrix. Hence, it is not straightforward to incorporate arbitrary graph indexes into their technique. In addition, our method for identifying substructures for indexing is different. In [8], it applies a path-decomposition method to partition a DAG into paths. Next, a path-path graph is proposed to capture the path relationship in a DAG and is indexed with interval labelings. Each node u in a DAG is assigned with an X label denoting the DFS order, a Y label denoting the path order and the interval labels I of u's corresponding path in the path-path graph. The reachability query between two nodes u and v can be answered by comparing their X, Y, and I labels. Although we are working on building graph framework through graph partitioning, our partitioning method is to decompose the input graph into arbitrary subgraphs. That is, the structure of each partition is more general than paths.

Graph partitioning has been one of the classical problems in combinatorial optimization. The problem optimizes an input objective function. In general, this is an NP-complete problem. Various heuristics, *e.g.*, [16,17,18], have been proposed to find an optimal partition, with respect to the objective function. In this paper, our objective function is different from those solved by previous algorithms.

6 Conclusions and Future Work

In this paper, we proposed a uniform framework for efficiently processing reachability query on large graphs. Specifically, a graph is represented by a set of partitions and inter-partition connections. Subsequently, (possibly different) graph indexes can be applied to each partition. This facilitates a seamless application of the state-of-the-art of graph indexing on subgraphs represented in the graph framework. Our experimental study verified the effectiveness and efficiency of our framework. In our experiment with a large variety of synthetic graphs and real graphs, our framework consistently produced relatively small indexes when compared to the best index of a non-partition approach. In addition, our experiment showed that the framework improves the query processing performance over the non-partitioning methods.

We would like to point out that our proposed method has some limitations. We plan to extend our work in the future: First, our framework is proposed to enhance reachability query performance. Yet, reachability queries can be a part of other query formalisms. We are studying the connection between reachability queries and other query formalisms. Second, our query evaluation algorithm is proposed to evaluate one query at a time on a single machine. We plan to study distributed reachability query evaluation on a graph framework.

Acknowledgements. We are grateful to Jiefeng Cheng and Dr. Ruoming Jin for providing us the implementation for 2-hop labelings and Path tree labeling. We thank R. Bramandia for his insightful discussions.

References

1. W3C: OWL web ontology language overview, `http://www.w3.org/TR/owl-features`
2. Wu, X., Lee, M.L., Hsu, W.: A prime number labeling scheme for dynamic ordered xml trees. In: ICDE, pp. 66–78 (2004)
3. Dietz, P.F.: Maintaining order in a linked list. In: STOC, pp. 122–127 (1982)
4. Chen, L., Gupta, A., Kurul, M.E.: Efficient algorithms for pattern matching on directed acyclic graphs. In: ICDE, pp. 384–385 (2005)
5. Wu, G., Zhang, K., Liu, C., Li, J.: Adapting prime number labeling scheme for directed acyclic graphs. In: Li Lee, M., Tan, K.-L., Wuwongse, V. (eds.) DASFAA 2006. LNCS, vol. 3882, pp. 787–796. Springer, Heidelberg (2006)
6. Chen, L., Gupta, A., Kurul, M.E.: Stack-based algorithms for pattern matching on dags. In: VLDB, pp. 493–504 (2005)
7. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. In: SIGMOD, pp. 253–262 (1989)
8. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficiently answering reachability queries on very large directed graphs. In: SIGMOD, pp. 595–608 (2008)
9. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM J. Comput. 32(5), 1338–1355 (2003)
10. Schenkel, R., Theobald, A., Weikum, G.: Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In: ICDE, pp. 360–371 (2005)
11. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: Answering graph reachability queries in constant time. In: ICDE, pp. 75–75 (2006)
12. Gibson, D., Kumar, R., Tomkins, A.: Discovering large dense subgraphs in massive graphs. In: VLDB, pp. 721–732 (2005)
13. Cheng, J., Yu, J.X., Lin, X., Wang, H., Yu, P.S.: Fast computation of reachability labeling for large graphs. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 961–979. Springer, Heidelberg (2006)
14. Trißl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: SIGMOD, pp. 845–856 (2007)
15. He, H., Wang, H., Yang, J., Yu, P.S.: Compact reachability labeling for graph-structured data. In: CIKM (2005)
16. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. The Bell system technical journal 49(1), 291–307 (1970)
17. Karypis lab: Family of Multilevel Partitioning Algorithms, `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`
18. Pothen, A., Simon, H.D., Liou, K.P.: Partitioning sparse matrices with eigenvectors of graphs. SIAM J. Matrix Anal. Appl. 11(3), 430–452 (1990)
19. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. 1(2), 146–160 (1972)
20. Schenkel, R., Theobald, A., Weikum, G.: Hopi: An efficient connection index for complex xml document collections. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., Ferrari, E. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 237–255. Springer, Heidelberg (2004)
21. Batagelj, V., Mrvar, A.: Pajek datasets, `http://vlado.fmf.uni-lj.si/pub/networks/data/`