Data Science and Engineering (2019) 4:193–207
https://doi.org/10.1007/s41019-019-00099-9

Modular Decomposition-Based Graph Compression for Fast Reachability Detection

Shikha Anirban[1] · Junhu Wang[1] · Md. Saiful Islam[1]

Received: 26 July 2019 / Revised: 19 August 2019 / Accepted: 21 August 2019 / Published online: 30 August 2019

Abstract
Fast reachability detection is one of the key problems in graph applications. Most of the existing works focus on creating an index and answering reachability based on that index. For these approaches, the index construction time and index size can become a concern for large graphs. More recently query-preserving graph compression has been proposed, and searching reachability over the compressed graph has been shown to be able to significantly improve query performance as well as reducing the index size. In this paper, we introduce a multilevel compression scheme for DAGs, which builds on existing compression schemes, but can further reduce the graph size for many real-world graphs. We propose an algorithm to answer reachability queries using the compressed graph. Extensive experiments with four existing state-of-the-art reachability algorithms and 12 real-world datasets demonstrate that our approach outperforms the existing methods. Experiments with synthetic datasets ensure the scalability of this approach. We also provide a discussion on possible compression for *k*-reachability.

Keywords Modular decomposition · Graph compression · Reachability queries · Algorithms

1 Introduction

The reachability query, which asks whether there exists a path from one vertex to another in a directed graph, finds numerous applications in graph and network analysis. Such queries can be answered by graph traversal using either breadth-first or depth-first search in time $O(|E| + |V|)$ without preprocessing (where V and E are the vertex set and edge set, respectively), or in constant time if we pre-compute and store the transitive closure of each vertex, which takes $O(|V||E|)$ time and $O(|V|^2)$ space. Unfortunately, neither of these approaches is feasible for applications that need to process large graphs with limited memory. Over the last decades, the problem has been extensively studied and many advanced algorithms have been proposed, with most of them relying on building smart indexes that can strike a balance between online query processing time and offline index construction time (and index size).

More recently, researchers recognized that it is possible to reduce the graph size by graph compression without loosing reachability information, and the compressed graph can help speedup query processing as well as reduce index size and index construction time. Specially, Fan et al. [7] define equivalence classes of vertices with respect to reachability queries, and compress a graph by merging all vertices in an equivalence class into a single vertex. However, finding all equivalence classes is very time-consuming. Zhou *et al* [22] propose an efficient algorithm to do a *transitive reduction* which turns a directed acyclic graph (DAG) into a DAG without redundant edges, after that the *equivalence reduction* of [7] can be done much more efficiently. The resulting graph G^{ϵ} after transitive reduction and equivalence reduction over the original graph G can be a much smaller graph that retains all reachability information, and it was experimentally verified that for many real-world graphs, searching for reachability over G^{ϵ} can be much faster than searching over G using state-of-the-art algorithms.

This paper builds on the work of [22]. We observe that after the removal of redundant edges, many *linear chains* will be generated. Based on this, we propose a multilevel reachability—preserving compression method that can

✉ Junhu Wang
j.wang@griffith.edu.au

Shikha Anirban
shikha.anirban@griffithuni.edu.au

Md. Saiful Islam
saiful.islam@griffith.edu.au

1 School of Information and Communication Technology, Griffith University, Gold Coast, Australia

further reduce the size of the graph obtained by the method in [22]. Our compression utilizes a slightly modified concept of *module* [14], and constructs a modular decomposition tree. We show how to use the decomposition tree to answer reachability queries over the original graph efficiently. Furthermore, the decomposition tree usually takes very small space. We make the following contributions:

1. We define a new concept of *module*, based on which we propose a multilevel graph compression scheme that compresses graphs into a smaller graph G_c.
2. We organize the modules into a hierarchical structure called *modular decomposition tree*, and propose an efficient algorithm to utilize the tree to answer reachability queries.
3. We conduct extensive experiments with real-world graphs as well as synthetic graphs that demonstrate the advantages of our proposed approach.
4. We provide a discussion on similar compression strategies for k-reachability queries.

The remainder of this paper is organized as follows. We first discuss related works in Sect. 2 and present the preliminaries in Sect. 3. Then, we give an overview of our approach and provide the theoretical foundations in Sect. 4, followed by the detailed algorithms in Sect. 5. Our experimental results are given in Sect. 6. Section 7 discusses the possibility of similar compression on k-reachability queries. We conclude our paper in Sect. 8.

2 Related Work

As briefly mentioned in Sect. 1, existing approaches for answering reachability queries can be classified into index-based and compression-based approaches.

2.1 Index-Based Approach

The index-based algorithms create labels for the vertices, and such labels contain the reachability information. These algorithms can be divided into *Label-only* and *Label+G* methods [19]. The label-only [1, 3, 4, 8, 10–13, 20] methods use the labels of source and destination vertices only to answer reachability. Agrawal [1] proposed tree cover approach that creates an optimal spanning tree to create index. Here, an interval for each vertex is created. A reachability query is answered as true if the interval of target is contained in the interval of source vertex. The index construction time and index size both are high in this approach. A *chain cover* approach is first proposed in [8] where the entire graph is divided into a number of pairwise disjoint chains to create the index. The label of each vertex contains a minimal successor list containing their chain number and position in the chain. A vertex u will be reachable to v if label of u contains a pair (k, j) and v has an index pair (i, j) such that $i \geq k$. This chain cover approach is later improved in [3]. Path tree [10] uses the similar concept of chain cover that uses paths to create index and has smaller index size than chain cover. The recent approaches DL [11], PLL [20] and TF [4] use the concept of two-hop labeling proposed in [6]. In two-hop labeling, a label is created for each vertex containing the subset of vertices that it can reach (L_{out}) as well as the subset of vertices that can reach it (L_{in}). Vertex u can reach vertex v if $L_{\text{out}}(u) \cap L_{\text{in}}(v) \neq \emptyset$. [12] uses the concept of chain cover to improve two-hop and proposes a three-hop labeling that creates a transitive closure contour ($Con(G)$) of graph G using chain decomposition, and then applies two-hop techniques. Path-hop [2] improves three-hop by replacing the chain decomposition with a spanning tree. TF [4] proposes a topological folding approach for two-hop labeling that can significantly reduce the index size as well as the query time.

The Label+G approaches include [15–19, 21] which require online searching of data graph G if the query cannot be answered from labels. Triβl and Leser [17] uses interval labeling over a spanning tree and performs DFS online if needed. Grail [21] and Ferrari [15] use multiple interval instead of single interval label for each vertex over the spanning tree. Feline [18] creates coordinates $i(u) = (X_u, Y_u)$ for a vertex u and answers reachability from u to v as true if the area of $i(v)$ is contained in that of $i(u)$. Feline also uses interval labeling over spanning tree and compares topological levels of u and v as additional pruning strategy to reduce DFS search. IP [19] uses independent permutation numbering to label each vertex. Feline and IP show significant improvement on query time and require less index construction time and smaller index size. BFL [16] proposes a bloom-filter labeling to further improve the performance of IP.

2.2 Compression-Based Approach

Graph compression-based works include scarab [9], equivalence reduction [7] and DAG reduction [22]. Scarab [9] compresses the original graph by creating a reachability backbone that carries the major reachability information. To find reachability from vertex u to vertex v, the algorithm needs access to a list of local outgoing backbone vertices of u and local incoming backbone vertices of v. The algorithm then performs a forward BFS for u and backward BFS for v on the original graph to answer reachability from u to v. If the answer is false, then it checks whether any outgoing backbone vertex of u can reach any incoming backbone vertex of v in the reachability backbone; if yes, then u can reach v. Scarab requires large index size with high time complexity. Equivalence reduction [7] reduces the graph by merging equivalent vertices into a single vertex. Two vertices are equivalent if they have the same ancestors and same descendants. The algorithm requires high equivalence class construction time. DAG reduction [22] improves the construction time of equivalence classes by doing a transitive reduction in graph first.

Our work is different from the previous works; in that, we not only consider equivalent classes, but also linear chains, when compressing the graph, and to the best of our knowledge, none of the previous works uses multilevel compression and modular decomposition tree in reachability queries.

3 Preliminaries

We consider directed graphs in this paper. For any directed graph G, we will use V_G and E_G to denote the vertex set and the edge set of G, respectively. Given two vertices u and v in G, if there is a path from u to v, we say v is *reachable* from u, or equivalently, u can *reach* v. We use $u \rightsquigarrow_G v$ to denote u can reach v in graph G. Given directed graph G and vertices u and v in G, a reachability query from u to v denoted $?u \rightsquigarrow_G v$, asks whether v is reachable from u in G.

A directed acyclic graph (DAG) is a directed graph without cycles. In the literature, most works on reachability queries assume the graph G is a DAG, because if it is not, it can be converted into a DAG by merging all vertices in a strongly connected component into a single vertex, and vertices in a strongly connected component can all reach each other. In this work, we also assume the graph G is a DAG.

If (u, v) is an edge in DAG G, we say u is a *parent* of v, and v is a *child* of u. For any vertex $u \in V_G$, we will use $parent(u, G)$ and $child(u, G)$, respectively, to denote the set of parents of u and the set of children of u in G. We will also use $anc(u, G)$ and $des(u, G)$ to denote the set of ancestors of u and the set of descendants of u in G, respectively. When G is clear from the context, we will use the abbreviations $parent(u)$, $child(u)$, $anc(u)$, and $des(u)$ for $parent(u, G)$, $child(u, G)$, $anc(u, G)$, and $des(u, G)$, respectively.

Let M be a subset of vertices in G. For any vertex $u \in M$ and a parent vertex u' of u, we say u' is an *external* parent of u (with respect to M) if $u' \in parent(u) - M$. Similarly, we define an *external* child (resp. ancestor, descendent) of u with respect to M as a vertex in $child(u) - M$ (resp. $anc(u) - M$, $des(u) - M$).

3.1 Redundant Edges

Suppose (u, v) is an edge in G. If there is a path of length greater than 1 from u to v, then (u, v) is *redundant* for reachability queries, that is, removing (u, v) from G will not affect the answer to any reachability queries.

The redundant edges can be efficiently identified and removed by a *transitive reduction* algorithm proposed in [22]. The following lemma is shown in [22]:

Lemma 1 *Suppose G is a DAG without redundant edges, then for any two vertices u and v in G,* $parent(u) = parent(v)$ *if and only if* $anc(u) = anc(v)$; $child(u) = child(v)$ *if and only if* $des(u) = des(v)$.

3.2 Equivalence Class

Two vertices u and v are said to be *equivalent* if they have the same ancestors and the same descendants, that is, $anc(u) = anc(v)$, $des(u) = des(v)$ [7]. Because of Lemma 1, if G does not have redundant edges, then u and v are equivalent if and only if they have the same parents and same children. The equivalent vertices form an equivalence class. It is easy to see that all vertices in the same equivalence class have the same reachability properties, that is, if u is in an equivalence class, then for any other vertex u', u can reach u' (resp. u is reachable from u') if and only if every vertex v in the same equivalence class can reach u' (resp. is reachable from u').

Also as observed in [22], if G has no redundant edges, then all vertices in an equivalence class form an independent set, that is, there are no edges between the vertices in the same equivalence class.

Lemma 2 *Suppose G is a DAG without redundant edges, then every equivalent class is an independent set.*

3.3 Modular Decomposition

The modular decomposition [14] of a directed graph G partitions the vertex set into a hierarchy of *modules*, where a module is conventionally defined as follows.

Definition 1 Let M be a set of vertices in G. We call M a *module* of G if all vertices in M share the same external parents and the same external children. In other words, for any $u, v \in M$, $parent(u) - M = parent(v) - M$ and $child(u) - M = child(v) - M$.

It is easy to see that a singleton set is a module and the set of all vertices in G is also a module. These modules are called *trivial* modules. Let G be a DAG that has no redundant edges. By Lemma 1, an equivalent class is also a module, and by Lemma 2, such a module is an independent set. In the literature, modules that are independent sets are referred to as *parallel* modules.

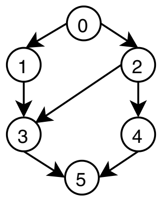


Fig. 1 Example DAG

4 Overview of Our Approach

The basic idea of our method is to compress the graph without loosing reachability information. We use *modular decomposition*; however, the definition of modules has been slightly modified from that found in the literature, in order to help with reachability queries.

Definition 2 A *module* in a DAG G is a set of vertices $M \subseteq V_G$ that have identical external ancestors and identical external descendants. In other words, for any two vertices $u, v \in M$, $anc(u) - M = anc(v) - M$, and $des(u) - M = des(v) - M$.

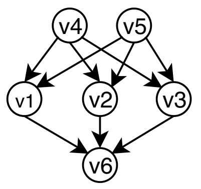
Figure 1 shows an example DAG, where the vertices 1, 2, 3, 4 have the same external ancestors and the same external descendants, but not the same external parents and same external children.

In this work, we are interested in two special types of modules, referred to as *parallel modules* and *linear modules*, respectively. A parallel module is a module that is an independent set, and a linear module is one that consists of a *chain* of vertices $v_1, \dots, v_k$ such that there is an edge (v_i, v_{i+1}) for all $i \in [1, k-1]$. These modules have the following properties.

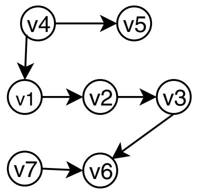
Lemma 3 *Suppose G is a DAG that does not have redundant edges. (1) If M is a parallel module of G, then all vertices in M have the same parents and same children. (2) If M is a linear module consisting of the chain $v_1, \dots, v_k$, then for each $i \in [2, k]$, v_{i-1} is the only parent of v_i, and v_i is the only child of v_{i-1}.*

Proof (1) Let M be a parallel module. By definition, M is an independent set, and all the vertices have the same external ancestors and the same external descendants. Since M is an independent set, it is impossible for any vertex in M to have an ancestor or descendent in M; therefore, all the vertices have the same ancestors and the same descendants (both external and internal). By Lemma 1, all vertices in M have the same parents and the same children.

(2) Let M be a linear module consisting of the chain $v_1, \dots, v_k$. For any $i \in [2, k]$, if v_i has a parent u that is not v_{i-1}, then there are two possible cases. The first case is that u is also in M, that is, u is one of $v_{i+1}, \dots, v_k$. This contradicts the assumption that G is a DAG since there will be a cycle. The second case is that u is not in M. In this case, by the definition of a module, u must be an ancestor of v_1, that is, there will be a path from u to v_i with length at least 2. Hence, the edge (u, v_i) would be redundant, contradicting the assumption that there are no redundant edges in G. This proves v_{i-1} is the only parent of v_i. Similarly, we can prove v_i is the only child of v_{i-1}. □



(a) $\{v1, v2, v3\}$ is a parallel module



(b) $\{v1, v2, v3\}$ is a linear module

Fig. 2 Example of modules

In Fig. 2a, the vertices $v1, v2, v3$ form a parallel module. In Fig. 2b, the vertices $v1, v2, v3$ form a linear module. Note, however, the set $\{v4, v1, v2, v3, v6\}$ in Fig. 2b is not a linear module.

It is worth noting that each single vertex forms a parallel module as well as a linear module. These modules are referred to as *trivial* modules, along with the module that consists of all of the vertices in G. According to Lemma 3, a parallel module is an equivalence class, if G is a DAG that has no redundant edges.

Note that if two vertices are in the same linear module, then their reachability depends on their relative positions in the chain. If they are in the same parallel module, then they cannot reach each other, as shown in the lemma below.

Lemma 4 *Let G be a DAG without redundant edges, and u, v be vertices in the same parallel module of G, then u cannot reach v in G.*

Proof Let the parallel module that contains u and v be M. If the lemma is not true, there will be a path $u, v_1, \dots, v_s, v$ from u to v. Since M is an independent set, v_1 and v_s cannot be in M. Hence, v_1 is an external child of u, and v_s is an external parent of v. By Lemma 3 and the definition of modules, v_1 must be a child of v and v_s must be a parent of u. Therefore, there will be a cycle, contradicting the assumption that G is a DAG. Hence, the proof. □

A set of vertices may be in multiple parallel (or linear) modules, e.g., in the graph shown in Fig. 2a, $\{v1, v2\}$ and $\{v1, v2, v3\}$ are both parallel modules. However, we are only interested in the *maximal* modules as defined below.

Definition 3 A parallel (resp. linear) module M is said to be *maximal* if there is no other parallel (resp. linear) module M' such that $M \subset M'$.

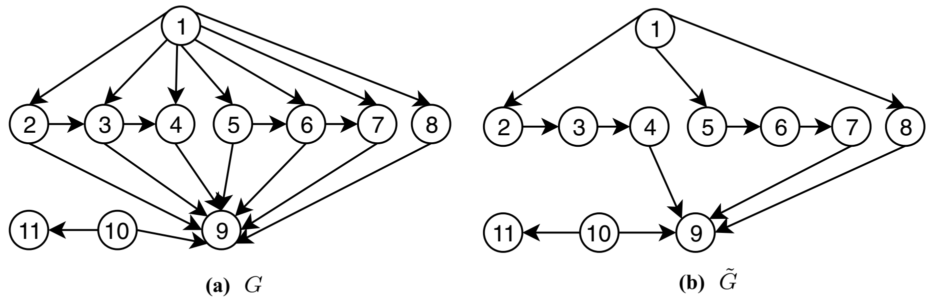


Fig. 3 **a** A DAG G and **b** the DAG $\tilde{G}$ after transitive reduction

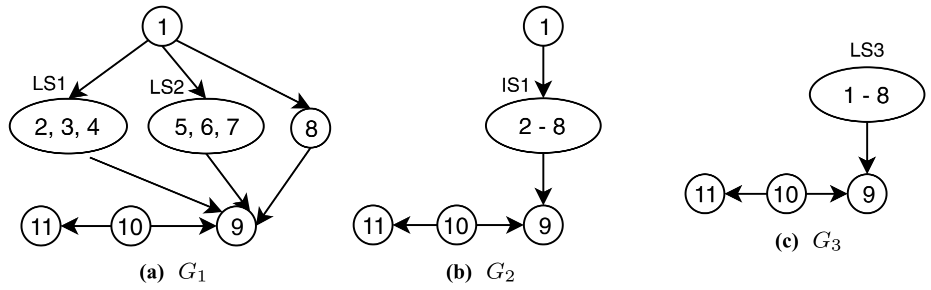


Fig. 4 **a** Graph G_1 after first-level compression, **b** graph G_2 after second-level compression and **c** graph G_3 after final compression. LS denotes linear module, and IS denotes parallel module

For example, $\{v1, v2, v3\}$ is a maximal parallel module in Fig. 2a, and it is a maximal linear module in Fig. 2b.

Note that two different maximal parallel modules of G cannot have overlaps, and two different maximal linear modules cannot have overlaps. Furthermore, there cannot exist a non-trivial parallel module and a non-trivial linear module such that they have a common vertex. In other words, each vertex can belong to at most one non-trivial parallel or linear module.

4.1 Multilevel Compression and Modular Decomposition Tree

To utilize parallel and linear modules in reachability search, we perform a *multilevel compression* of the original graph G. First, we identify the maximal linear modules and parallel modules, and merge the vertices in each module into a single super vertex. We add an edge from super vertex s_1 to super vertex s_2 if and only if there exists $u \in s_1$, and $v \in s_2$ such that (u, v) is an edge in G. In this way, we obtain the first-level compressed graph $G_1 = Compress(G)$. Clearly, G_1 is also a DAG without redundant edges. Then, we apply the same compression process to G_1 to obtain the next-level compressed graph $G_2 = Compress(G_1)$, and this process is repeated until we obtain a graph G_c which can no longer be compressed, i.e., G_c does not have singleton set parallel or linear modules.

Example 1 Consider the DAG G in Fig. 3a, which consists of eleven vertices numbered 1 to 11. The graph is reduced to $\tilde{G}$ in Fig. 3b after transitive reduction. We will apply our compression to graph $\tilde{G}$.

There are no parallel modules in $\tilde{G}$. However, vertices 2, 3 and 4 can form a maximal linear module. Another maximal linear module exists in $\tilde{G}$ that consists of vertices 5, 6 and 7. So, vertices 2, 3, 4 and vertices 5, 6, 7 are compressed into two single nodes, and they are reduced into nodes LS1 and LS2, respectively, in graph G_1 shown in Fig. 4a after the first-level compression. Then, G_1 is compressed again to obtain G_2 as shown in Fig. 4b, where the nodes LS1, LS2 and 8 in G_1 are merged as they form an equivalent set in G_1. The third-level compression creates graph G_3 in Fig. 4c by merging nodes 1 and IS1 in G_2 which form a linear module. The graph G_3 does not contain any parallel or linear modules thus cannot be compressed further. So, G_3 is the final compressed graph of data graph G.

We organize the modules in all levels of the compressed graphs into a tree structure, called the *modular decomposition tree*, or *decomposition tree* for brevity, as follows: The root of the tree is the final compressed graph G_c. Each module in the previous-level compressed graph G_{c-1} is a child node of the root; Each child node of the root that corresponds to a non-trivial module of G_{c-1}, in turn, has its own children, representing modules in the previous-level graph G_{c-2}. This continues until we reach the nodes representing modules in the first-level compressed graph, where each non-trivial module points to their children, which are individual vertices in the original graph G. Note that the leaf nodes of the tree are individual vertices in the original graph G. Also, to help reachability detection, we keep a record of the vertex positions in the chain of each linear module in a

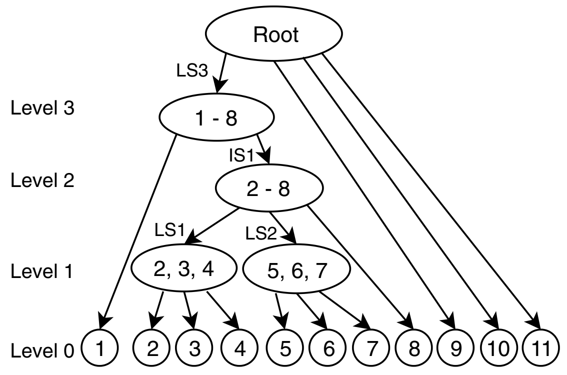


Fig. 5 The modular decomposition tree T of graph $\tilde{G}$

compressed graph G_i, where if the starting vertex has position 1, the next vertex will have position 2 and so on. We will use $pos(v, LS)$ to denote the position of node v in the chain of LS. Obviously, for $u, v \in LS$, $u \rightsquigarrow_{G_i} v$ if and only if $pos(u, LS) < pos(v, LS)$.

Figure 5 shows the modular decomposition tree, T, of graph $\tilde{G}$ in Fig. 3b. Let M be a non-leaf node in the decomposition tree of G. By definition, M is either a parallel or linear module in some compressed graph G_i $(i < c)$, or it is the final compressed graph G_c. M can be regarded as a set of the original vertices of G in the obvious way. Put in another way, we say vertex $v \in G$ belongs to (or is in) M if v is a descendant of M in the decomposition tree. For example, the vertices 2, 3, 4, 5, 6, 7, 8 belong to the module IS1 in Fig. 5.

We have the following observations about modules in the decomposition tree:

Lemma 5 *The vertices of G that belong to each parallel or linear module M in G_i $(i < c)$ form a module of G. In other words, all vertices in M have the same external ancestors, as well as the same external descendants in G.*

The above lemma can be easily proved by induction on the compression level i. Using Lemma 5, we can easily see:

Lemma 6 *Given two distinct nodes N_1 and N_2 in G_i $(i \leq c)$, $N_1 \rightsquigarrow_{G_i} N_2$ iff $u \rightsquigarrow_G v$ for every pair of vertices $u \in N_1$, $v \in N_2$.*

4.2 Answering Reachability Queries Using Modular Decomposition Tree

Suppose we have the decomposition tree T of G. For ease of presentation, let us use G_0 to denote the graph G. For any pair of vertices u, v in the original graph G, we use $LCA(u, v)$ to denote the lowest common ancestor of u and v in T. Note that $LCA(u, v)$ corresponds either to a module in some compressed graph G_i $(i \in [1, c-1])$, or to the final compressed graph G_c (i.e., the root of T). We have the following result.

Theorem 1 *Given two vertices $u, v \in V_G$, if $LCA(u, v)$ corresponds to a parallel module of some graph G_i $(i \in [1, c-1])$, then u cannot reach v in G.*

Proof If $LCA(u, v)$ corresponds to a parallel module M of G_i, then suppose N_1 and N_2 are the two vertices of G_i that contain u and v, respectively. By Lemma 4, we know N_1 cannot reach N_2 in G_i. Then by Lemma 6, we know u cannot reach v in G. □

With the above discussion, we are ready to present the method for answering reachability queries using the decomposition tree and the final compressed graph G_c. Given two vertices $u, v \in V_G$, to find whether $u \rightsquigarrow_G v$, we can find the lowest common ancestor $LCA(u, v)$ of u and v, and check the following:

1. If $LCA(u, v)$ is a parallel module, then u cannot reach v in G, by Theorem 1.
2. If $LCA(u, v)$ is a linear module, say M, then we check the positions of N_1 and N_2 in the corresponding chain of vertices in M, where N_1 is the child of $LCA(u, v)$ in the decomposition tree that contains u, and N_2 is the child of $LCA(u, v)$ that contains v, and u can reach v in G if and only if $pos(N_1, M) < pos(N_2, M)$.
3. If $LCA(u, v)$ is the root of T, namely G_c, then suppose N_1, N_2 are the children of $LCA(u, v)$ that contain u and v, respectively. Then, $u \rightsquigarrow_G v$ if and only if $N_1 \rightsquigarrow_{G_c} N_2$. Thus, we only need to check whether $N_1 \rightsquigarrow_{G_c} N_2$. We can do it using any existing reachability algorithms. Since G_c is usually much smaller than G, checking $N_1 \rightsquigarrow_{G_c} N_2$ in G_c is likely to be faster than checking $u \rightsquigarrow_G v$ in G.

Example 2 Consider the decomposition tree T shown in Fig. 5.

(1) For the query $?2 \rightsquigarrow_G 6$, we find that lowest common ancestor of vertices 2 and 6 is a parallel module; therefore, we know vertex 2 cannot reach vertex 6.
(2) For the query $?2 \rightsquigarrow_G 4$, we find $LCA(2, 4)$ is a linear module, and the position of vertex 2 is before that of vertex 4. Therefore, we conclude that $2 \rightsquigarrow_G 4$.
(3) For the query $?2 \rightsquigarrow_G 9$, since 2 and 9 are in different children of the root, i.e., $LS3$ and 9, respectively, we only need to check whether $LS3 \rightsquigarrow_{G_3} 9$.

5 Algorithms

The previous section provides the main ideas of our approach. This section presents the detailed algorithms.

5.1 Building Modular Decomposition Tree

Algorithm 1 shows the process of creating the modular decomposition tree along with the final compressed graph. The algorithm takes a DAG that has no redundant edges G as input and returns the modular decomposition tree and the final compressed graph. The algorithm first creates a tree with a root node r. Starting with a random vertex v, the algorithm first tries to find all other vertices that can form a linear module with v (Line 7). If no such module is found, then it will search for a maximal parallel module for v (Line 14). If such a module cannot be found, then v will be added as a child of r (Line 21), otherwise the found module M will be added as child of r, and each vertex in the module will be added as a child of M (Lines 8–12, 16–19). We record all such modules in S (Lines 9,16), and use them to compress the graph into a new graph (Line 24). Then, we recursively call the algorithm to compress the new graph (Line 27). If no non-single-vertex module is found in the current graph, the current tree T will be returned, and the current graph will be returned as G_c.

Algorithm 1: BuildMDT(G)

```
Input: DAG $G$ with no redundant edges
Output: Modular Decomposition Tree $T$ and $G_c$
1  if $T$ does not exist then
2      Create Tree $T$ with root node $r$;  $i \leftarrow 0$;  $G_i \leftarrow G$
3  $S \leftarrow \emptyset$
4  for each $v \in V_{G_i}$ do
5      if $v.isVisited$ is false then
6          $v$.isVisited $\leftarrow$ true
7          $M \leftarrow FindLinearModule(v, G_i)$
8          if $M \neq null$ then
9              $S \leftarrow S \cup M$
10             Add $M$ as child of $r$
11             for each vertex $u$ in $M$ do
12                 $u$.isVisited $\leftarrow$ true;  Add $u$ as a child of $M$
13         else
14             $M \leftarrow FindParallelModule(v, G_i)$
15             if $M \neq null$ then
16                 $S \leftarrow S \cup M$
17                 Add $M$ as child of $r$
18                 for each vertex $u$ in $M$ do
19                     $u$.isVisited $\leftarrow$ true;  Add $u$ as a child of $M$
20             else
21                 Add $v$ as a child of $r$
22 if $S \neq \emptyset$ then
23     i++
24     $G_i \leftarrow Compress(G_{i-1}, S)$
25     BuildMDT($G_i$)
26 else
27     $r \leftarrow G_i$
28     return $T$, $G_i$
```

Algorithm 2: FindParallelModule

```
Input: DAG $G$ with no redundant edges, vertex $v$
Output: The maximal nontrivial parallel module that $v$ is in, or null if such module
        does not exist
1  Create module $M = \{v\}$
2  $M.type = trivial$
3  if $|parent(v)| = 0$ then
4      $M_1 \leftarrow \{v' |\ |parent(v')| = 0\}$
5  else
6      $M_1 \leftarrow \{v' | v' \in \bigcap_{u \in parent(v)} child(u), v' \neq v \ and \ |parent(v)| = |parent(v')|\}$
7  if $|child(v)| = 0$ then
8      $M_2 \leftarrow \{v' |\ |child(v')| = 0\}$
9  else
10     $M_2 \leftarrow \{v' | v' \in \bigcap_{u \in child(v)} parent(u), v' \neq v \ and \ |child(v)| = |child(v')|\}$
11 if $M_1 \cap M_2 \neq \emptyset$ then
12     $M.type \leftarrow Parallel$ $M \leftarrow (M_1 \cap M_2) \cup M$ Return $M$
13 else
14     Return null
```

The functions FindParallelModule() and FindLinearModule() used in Algorithm 1 are shown in Algorithm 2 and Algorithms 3, respectively. These algorithms try to find a relevant module based on Lemma 3.

Algorithm 2 takes a vertex v and DAG G as input, and it finds the set of vertices, M, that share the same parents and same children with v. To do that, it first finds the set of vertices, M_1 that share the same parents with v, and then finds the set of vertices, M_2 that share the same children with v. Then, M is the intersection of M_1 and M_2.

Algorithm 3: FindLinearModule

```
   Input: DAG $G$ with no redundant edges, vertex $v \in V_G$
   Output: The maximal nontrivial linear module that $v$ is in, or null if such module does
           not exist
1  Create module $M = \{\}$;  $M.type = trivial$
2  if $|parent(v)| = 1$ then
3      $v' \leftarrow$ unique parent of $v$
4      if $|child(v')| = 1$        /* $v'$ and $v$ are in the same linear module
5       then
6          add $v, v'$ to $M$;  $M.type \leftarrow$
             $Linear$;  $pos(v, M) \leftarrow 1$;  $pos(v', M) \leftarrow pos(v, M) - 1$
7          while $|parent(v')| = 1$ do
8              $u \leftarrow$ unique parent of $v'$
9              if $|child(u)| = 1$ then
10                 add $u$ to $M$;  $pos(u, M) \leftarrow pos(v', M) - 1$;  $v' \leftarrow u$
11             else
12                 break
13 if $|child(v)| = 1$ then
14     $v' \leftarrow$ unique child of $v$
15     if $|parent(v')| = 1$ then
16         if $M.type = trivial$ then
17             add $v, v'$ to $M$;  $M.type \leftarrow Linear$
18             $pos(v, M) \leftarrow 1$;  $pos(v', M) \leftarrow pos(v, M) + 1$
19         else
20             add $v'$ to $M$;  $pos(v', M) \leftarrow pos(v, M) + 1$
21         while $|child(v')| = 1$ do
22             $u \leftarrow$ unique child of $v'$
23             if $|parent(u)| = 1$ then
24                 add $u$ to $M$;  $pos(u, M) \leftarrow pos(v', M) + 1$;  $v' \leftarrow u$
25             else
26                 break
27 if $M.type = trivial$ then
28     Return null
29 else
30     Return $M$
```

Algorithms 3 takes a vertex v and DAG G as input, and it first searches for a possible chain of ancestors of v (lines 2–12), and then searches for a chain of descendants of v (lines 13–26). Both parts are via an iterative process. Specifically, lines 2 and 4 check whether v has a sole parent v', and v' has a sole child v, if so v' is the parent of v in a linear module. After that, lines 7–12 try to find a parent of the first vertex in the chain, one by one. In the process, it also provides a position number for each vertex found (line 10). Note that the position does not have to be a positive number, as long as it can provide an appropriate order of the vertices in the chain, it will be fine. Lines 13–26 work similarly.

5.1.1 Complexity

Algorithm 3 takes L steps to find the linear module that contains v (checking the $|parent(|v|) = 1$ is just checking the in-degree of v), where L is the size of the linear module that contains v. Algorithm 2 takes $\Sigma_{u \in parent(v)} |child(u)| + \Sigma_{u \in child(v)} |parent(u)|$ steps to find vertices that share the same parents and same children with v. If we use $I_{\max}$ and $O_{\max}$ to denote the maximum in-degree and maximum out-degree, respectively, then Algorithm 2 takes $O(I_{\max} \times O_{\max})$ time. In Algorithm 1, for the first-level compression, we visit each vertex in V_G that has not been put in a module, and once the vertex is visited or put into a module, it will no longer be visited. In the worst case, no non-trivial module exists, so that every vertex will be visited. Therefore, the first-level compression takes $O(|V| \times I_{\max} \times O_{\max})$. Each next-level compression will take no more than that of the previous level. Therefore, Algorithm 1 takes $O(|V| \times I_{\max} \times O_{\max} \times h)$, where h is the height of the decomposition tree. In practice, h is usually small. For instance, in our experiment with 12 datasets, the highest decomposition tree has a height of 26 only.

Algorithm 4: Find reachability from vertex u to vertex v

```
Input: Modular decomposition tree $T$, Compressed Graph $G_c$, vertex u, vertex v
Output: true if $u$ is reachable to $v$, false otherwise
1  $N_1 \leftarrow$ Corresponding node of $u$ in $G_c$
2  $N_2 \leftarrow$ Corresponding node of $v$ in $G_c$
3  if $N_1 = N_2$ then
4      $M \leftarrow FindLinearLCA(u, v)$
5      if $M$ exists then
6          if $pos(u, M) < pos(v, M)$ then
7              return true
8      return false
9  else
10     return AlgoReachability($G_c, N_1, N_2$)
```

5.2 Finding Reachability Using the Decomposition Tree

As discussed in the previous subsection, to answer reachability query $?u \rightsquigarrow_G v$ using the decomposition tree, we only need to find $LCA(u, v)$ and then take appropriate actions depending on the type of $LCA(u, v)$. To save time for finding the LCA and storage space, we design a slightly modified algorithms as shown in Algorithm 4. Given vertices u and v, we first find the children of the root that u and v belong to, respectively; let us suppose $u \in N_1$, $v \in N_2$, if they are different (this is equivalent to say $LCA(u, v)$ is the root), we will use some existing algorithm to check $?N_1 \rightsquigarrow_{G_c} N_2$. Otherwise we will find the lowest *linear* LCA of u, v (note that to do so we only need to record the linear module ancestors of the vertices), if no such LCA exists, then u cannot reach v. Otherwise, suppose the linear LCA is M, we will check the relative positions of u and v in M to determine whether u can reach v. Here, the position of u is defined to be same as the position of the child of M that contains u. If $LCA(u, v)$ is a parallel module, then that module will be a child of M. So, u and v will have the same position in M thus u cannot reach v. It is easy to see that the algorithm is equivalent to the process described in Sect. 4; hence, its correctness is guaranteed.

5.2.1 Size of the Decomposition Tree

To answer reachability queries in the original graph G, we only need the final compressed graph G_c and the decomposition tree T. The total number of nodes in the tree is $|V| + m + 1$, where m denotes the number of non-trivial modules. The number of edges in T is $|V| + m$. The tree T can be regarded as an additional index. As shown in Algorithm 4, in implementation, we do not need to store the entire modular decomposition tree, instead, we only need to store the sequence of linear modules and the child module of the root (i.e., the node in G_c) each vertex belongs to.

6 Experiments

In this section, we present our experimental results. We compare our compression scheme and DAG reduction [22] on the performance of four state-of-the-art reachability query algorithms: Grail [21], Feline [18], IP$^+$ [19] and BFL$^+$ [16], which include query time and index size/construction time. We also report the compression time of our approach.

6.1 Experimental Setup

6.1.1 Implementation and Running Environment

We obtained the source code of DAG reduction, Grail, IP$^+$, Feline and BFL$^+$ from the authors which are all written in C++. We implemented our reachability query processing

Table 1 Real-world datasets and their compression ratio after DAG reduction and multilevel compression

Dataset	G		$\tilde{G}$	G^{ϵ}		G_c	
	|V|	|E|	r_e%	r_n%	r_e%	r_n%	r_e%
Kegg	3617	3908	93.8	37.6	35.7	9.7	9.3
arXiv	6000	66,707	20	97.9	19.7	93.3	19.3
XMark	6080	7025	99	55.8	57	25.7	31
PubMed	9000	40,028	67.5	76.7	62	76.2	61.9
soc-Epinions	42,176	43,797	96.6	19.9	19.3	13	12.7
Web	371,764	517,805	79.8	30.5	24.9	16.6	14.6
LJ	971,232	1,024,140	95.1	11.1	10.8	7.9	7.6
Patent	3,774,768	16,518,947	71.6	91.2	68.9	90.5	68.7
05Patent	1,671,488	3,303,789	90.1	80.3	78.9	78.8	78.2
05Citeseerx	1,457,057	3,002,252	81	37.9	50	37.4	49.7
Citeseerx	6,540,401	15,011,260	74.4	39.7	46.4	38.9	46.1
DBpedia	3,365,623	7,989,191	59.2	50.5	31.7	43.9	28.9

$r_n(r_e)$ is the ratio of the number of vertices (edges) in $\tilde{G}$, G^{ϵ} and G_c

Table 2 Graph size before and after compression

Dataset	G	G^{ϵ}		G_c	
	$\|V\| + \|E\|$	$\|V_{G^{\epsilon}}\| + \|E_{G^{\epsilon}}\|$	$r_{G^{\epsilon}}$%	$\|V_{G_c}\| + \|E_{G_c}\|$	r_{G_c}%
Kegg	7825	2756	35.2	720	9.2
arXiv	72,707	19,046	26.2	18,481	25.4
XMark	13,105	7394	56.4	3732	28.5
PubMed	49,028	31,730	64.7	31,632	64.5
soc-Epinions	85,973	16,846	19.6	11,016	12.8
Web	889,569	242,305	27.2	137,110	15.4
LJ	1,995,372	218,608	10.9	153,860	7.7
Patent	20,293,715	14,827,554	73.1	14,779,167	72.8
05Patent	4,975,277	3,949,609	79.4	3,901,493	78.4
05Citeseerx	4,459,309	2,052,235	46	2,037,926	45.7
Citeseerx	21,551,661	9,562,970	44.4	9,458,669	43.9
DBpedia	11,354,814	4,233,784	37.3	3,787,744	33.4

algorithms connecting with Grail, IP$^+$, Feline and BFI$^+$ in C++ using G++ 7.3.0 compiler. Our multilevel compression algorithm was implemented in C# using Visual Studio 2017. (since compression is done offline.) The experiments were run on a PC with Intel Core i7-7700 with 3.60 GHz CPU, 32 GB memory and Windows 10 operating system.

6.1.2 Datasets and Queries

We tested our approach with 12 real datasets and eight synthetic datasets. For each dataset, we first applied the transitive reduction in [22] to find $\tilde{G}$, which is a DAG without redundant edges. Then, we applied our multilevel compression algorithm to get G_c, and used DAG reduction to generate G^{ϵ}. We used Grail, IP$^+$, Feline and BFL$^+$ to process reachability queries over G_c and over G^{ϵ}. We randomly generated 100,000 reachability queries for each data graph, and each query was run ten times using our compression schema and that of [22], and the average time is recorded.

6.2 Experiments on Real Datasets

6.2.1 Datasets

We used 12 real datasets Kegg[1], arXiv[1], XMark[1], PubMed[1], Patent[1], Citeseerx,[1] soc-Epinions[2], Web,[2] LJ,[2] 05Patent[3], 05Citeseerx[3] and DBpedia.[4] Among these datasets, Kegg and XMark are very small graphs. Datasets arXiv, PubMed, soc-Epinions, Web and LJ are of medium size, whereas the other five graphs can be considered as large graphs. Here, Kegg is a metabolic network, and XMark is an XML document. Datasets soc-Epinions and LJ are the online social networks. Web is the web graph from Google. arXiv, PubMed, Patent, 05Patent, 05Citeseerx and Citeseerx are all citation networks, and DBpedia is a knowledge graph. The statistics of these datasets are shown in the first two columns of Table 1.

6.2.2 Compression Ratio

The compression ratios of transitive reduction, DAG reduction (i.e., transitive reduction and equivalence reduction) and our multilevel compression are shown in Table 1. From the table, we can see that our approach has more compression

[1] https://code.google.com/archive/p/grail/downloads.
[2] http://snap.stanford.edu/data/index.html.
[3] http://pan.baidu.com/s/1bpHkFJx.
[4] http://pan.baidu.com/s/1c00Jq5E.

Table 3 Size and storage space for decomposition tree

Dataset	Tree size $(\lvert V\rvert + \lvert m\rvert + 1)$	Space (MB) for decomposition tree	Space (MB) for equivalence class
Kegg	4238	0.009	0.006
arXiv	6352	0.01	0.01
XMark	8631	0.02	0.01
PubMed	9385	0.02	0.02
soc-Epinions	44,590	0.09	0.08
Web	427,907	0.9	0.7
LJ	997,681	1.93	1.85
Patent	3,981,729	7.3	7.2
05Patent	1,856,796	3.31	3.19
05Citeseerx	1,592,636	2.86	2.78
Citeseerx	7,139,965	12.99	12.47
DBpedia	3,804,103	7.49	6.41

for every graph than DAG reduction. The dataset XMark has the best result with 30.1% more compression of vertex and 26% more compression of edges than DAG reduction. For larger graphs, DBpedia shows best compression with 6.6% more compression of nodes and 2.8% more compression of edges. On the other hand, our compression scheme is only slightly better than DAG reduction over the Citeseerx and the Patent datasets. This could be because these datasets do not contain many linear modules. Generally, the reduction ratio depends on the structure of the graph. However, a small percentage of compression for a large graph can also have great impact on query processing since even a small percentage of compression means reduction in lots of vertices and edges in large graphs (see the Patent dataset in Table 7 for example).

Table 2 shows the size of G, G^{ϵ} and G_c. Here, we calculated the size of the graph as the sum of the number of vertices and the number of edges.

The second column of Table 3 shows the number of nodes in the modular decomposition tree for each of the data graph, which is calculated as $\lvert V\rvert + \lvert m\rvert + 1$ where $\lvert V\rvert$ is the number of vertices in the data graph which represents the leaf nodes in the tree and m is the number of non-trivial modules in the tree. The number of edges is $\lvert V\rvert + \lvert m\rvert$. As discussed in Sect. 5.2, we do not need to store the entire decomposition tree, we only need to store, for each vertex, its linear module ancestors and corresponding node in G_c. The required storage space is shown in the third column. For DAG reduction, it also needs to store the equivalence classes each vertex is in. The storage size is shown in the fourth column. As can be seen, our approach needs more storage space, but the difference is small. If we add this space and index size (see Table 6) together, our approach needs less overall space.

Table 4 Compression time (s)

Dataset	Time (s)
Kegg	0.057
arXiv	0.16
XMark	0.16
PubMed	0.49
soc-Epinions	4.49
Web	286.67
LJ	3073
Patent	389.23
05Patent	71.65
05Citeseerx	175.69
Citeseerx	8772.81
DBpedia	89,764.32

Table 4 shows the time required for building the decomposition tree using our algorithms which are implemented in C#, where the dataset DBpedia has taken the most time. As the indexing is done offline, we consider these times as viable in practice.

6.2.3 Index Construction Time

Table 5 shows the comparison of index construction time for Grail, IP^{+}, Feline and BFL^{+} algorithms over G^{ϵ} and G_c. The better results are highlighted in bold font in the table. Here, multilevel compression requires less index construction time for every graph for creating index for IP^{+} and BFL^{+}. For Feline, we also have better result for each graph except 05Patent. Grail performs better in multilevel compression for every graph except 05Citeseerx and Citeseerx.

6.2.4 Index Size

The index size of Grail, IP^{+}, Feline and BFL^{+} for G^{ϵ} and G_c are shown in Table 6. From the table, we can see that the index sizes of G_c are smaller for almost every graph than G^{ϵ} for all of the four algorithms, although for the arXiv, PubMed, Citeseerx and Patent datasets the difference is very small and for PubMed the index size of Feline is smaller in G^{ϵ} than in G_c as well. This is not surprising because the sizes of G_c and G^{ϵ} are very close for these datasets.

6.2.5 Query Performance

Table 7 shows the comparison of the query time for Grail, IP^{+}, Feline and BFL^{+}. We run each query ten times and the time shown is the average of the 10 runs. We can see that our compression outperforms DAG reduction in query processing for almost every graph. Surprisingly IP^{+} is lower using our approach than using DAG reduction in Kegg dataset; Feline is lower in LJ and BFL^{+} is lower in patent. For all

Table 5 Index construction time (ms)

Dataset	Grail		IP$^+$		Feline		BFL$^+$	
	G^ϵ	G_c	G^ϵ	G_c	G^ϵ	G_c	G^ϵ	G_c
Kegg	1.01	**0.64**	0	0	0.47	**0.25**	0.07	**0.02**
arXiv	4.03	**5.34**	0.015	**0**	2.03	**1.96**	0.66	**0.55**
XMark	2.03	**1.04**	0.02	**0**	1.5	**0.7**	0.14	**0.07**
PubMed	6.12	**5.05**	0.02	**0.01**	2.49	**2.25**	0.84	**0.75**
soc-Epinions	4.75	**3.13**	0.04	**0.01**	2.39	**1.02**	0.49	**0.24**
Web	71.39	**43.24**	0.04	**0.03**	46.21	**27.77**	9.52	**5.81**
LJ	64.59	**48.58**	0.06	**0.03**	39.08	**27.58**	7.52	**4.52**
Patent	5367.76	**5283.97**	5.6	**5.06**	3496.5	**3477.12**	1169.05	**1161.4**
05Patent	1407.6	**1408.6**	1.42	**1.25**	**950.03**	959.18	306.06	**301.48**
05Citeseerx	**516.78**	519.09	0.53	**0.51**	315.52	**314.87**	104.22	**102.61**
Citeseerx	**2697.92**	2711.95	3.26	**2.53**	1634.8	**1629.01**	757.73	**738.46**
DBpedia	1478.23	**1221.09**	1.23	**1.07**	937.71	**834.48**	256.32	**237.09**

Table 6 Index size (MB)

Dataset	Grail		IP$^+$		Feline		BFL$^+$	
	G^ϵ	G_c	G^ϵ	G_c	G^ϵ	G_c	G^ϵ	G_c
Kegg	0.005	**0.001**	0.04	**0.008**	0.03	**0.008**	0.05	**0.01**
arXiv	0.02	0.02	0.21	**0.2**	0.13	**0.12**	0.24	**0.23**
XMark	0.01	**0.006**	0.12	**0.05**	0.08	**0.03**	0.14	**0.06**
PubMed	0.03	**0.02**	0.19	0.19	**0.03**	0.15	0.22	**0.2**
soc-Epinions	0.03	**0.02**	0.21	**0.12**	0.19	**0.12**	0.27	**0.16**
Web	0.43	**0.23**	3.25	**1.62**	2.59	**1.41**	4.04	**2.01**
LJ	0.41	**0.29**	2.71	**1.72**	2.47	**1.75**	3.39	**2.16**
Patent	13.12	**13.03**	67.91	**66.84**	78.76	**78.22**	123.65	**122.6**
05Patent	5.12	**5.02**	18.35	**17.79**	30.71	**30.16**	42.43	**41.39**
05Citeseerx	2.1	**2.07**	12.64	**12.47**	12.63	**12.47**	17.08	**16.77**
Citeseerx	9.91	**9.71**	67.91	**66.84**	59.46	**58.27**	76.06	**73.7**
DBpedia	6.48	**5.64**	45.35	**38.54**	38.87	**33.84**	56.45	**47.62**

Table 7 Query time (ms)

Dataset	Grail		IP$^+$		Feline		BFL$^+$	
	G^ϵ	G_c	G^ϵ	G_c	G^ϵ	G_c	G^ϵ	G_c
Kegg	22.97	**19.89**	**26.96**	32.7	22.43	**18.11**	32.38	**25.49**
arXiv	123.68	**94.42**	82.25	**76.36**	96.07	**94.14**	61.95	**57.57**
XMark	25.86	**22.9**	32.39	**30.95**	29.29	**23.93**	33.78	**31**
PubMed	53.22	**50.15**	39.56	**37.92**	43.39	**40.73**	47.8	**42.74**
soc-Epinions	40.29	**35.5**	50.85	**45.26**	35.04	**28.13**	47.94	**45.54**
Web	96.93	**82.37**	145.6	**132.89**	75.14	**66.05**	117.86	**99.88**
LJ	120.41	**93.93**	144.27	**120.55**	**92.79**	95.45	161.85	**127.46**
Patent	831.7	**688.4**	375.74	**361.91**	592.53	**484.85**	**193.18**	199.2
05Patent	197.17	**159.57**	145.28	**139.81**	148.81	**146.47**	146.25	**143.71**
05Citeseerx	169.84	**166.59**	152.91	**140.41**	156.76	**152.27**	148.68	**145.53**
Citeseerx	208.44	**206.52**	223.37	**191.43**	200.95	**196.69**	232.57	**206.14**
DBpedia	216.21	**215**	221.47	**206.32**	206.32	**200.32**	215.62	**186.22**

Table 8 Datasets and their compression ratio after ER reduction and multilevel compression along with the compression time for multilevel compression for synthetic datasets

Dataset	|V|	|E|	r_n (%)		r_e (%)		Compression time (s)
			G^ϵ	G_c	G^ϵ	G_c	G_c
Data1	1000	2000	99.4	90.3	82.95	78.4	0.02
Data2	5000	8000	98.46	82.26	75.05	64.93	0.09
Data3	10,000	25,000	99.78	95.31	90.63	88.84	0.21
Data4	50,000	75,000	98.26	80.32	74.53	62.57	1.10
Data5	100,000	150,000	98.18	80.2	74.8	62.82	2.68
Data6	1,000,000	1,500,000	98.18	80.24	74.44	62.49	40.01
Data7	5,000,000	7,500,000	98.2	80.32	74.48	62.56	433.98
Data8	10,000,000	15,000,000	98.2	80.32	74.49	62.57	12,626.34

other datasets, the performance of multilevel compression is much better than DAG reduction.

6.3 Experiments on Synthetic Datasets

We generated eight random graphs where the smallest graph have one thousand vertices with two thousands edges, and the largest graph have ten millions vertices with 15 millions edges. The other graphs have number of vertices and edges in between these two graphs.

Table 8 shows the graph profiles and their compression ratio for DAG reduction and multilevel compression. The table also shows the time required for compression of each of the graphs using multilevel compression. Multilevel compression can process the largest graph with ten millions vertices and 15 millions edges within few hours which ensures the scalability of this method.

6.3.1 Compression Ratio

Figure 6 shows the comparison of compression ratio between DAG reduction and multilevel compression. We can see from figure that our method do more than 15% more compression on vertex and more than 10% more compression on edges for each of the graph except Data1 and Data3. The compression ratio of Data1 and Data3 is also better in multilevel compression.

6.3.2 Index Construction Time

We tested index construction time for the three reachability algorithms IP$^+$, Feline, and BFL$^+$. For Grail, we experienced errors; therefore, we omit it here. Figure 7 shows the index construction time for each of the algorithm for both G^ϵ and G_c. For index constructions that cannot be completed, we omit them from the figure. We can see from figure that the index construction time is much better for multilevel compression for each of the three algorithm and all datasets except Data5 for Feline algorithm.

6.3.3 Index Size

Figure 8 shows index size comparison for IP$^+$, Feline and BFL$^+$. The index size is much smaller for G_c than G^ϵ for each graph in all three algorithms. It is obvious as G_c is smaller than G^ϵ.

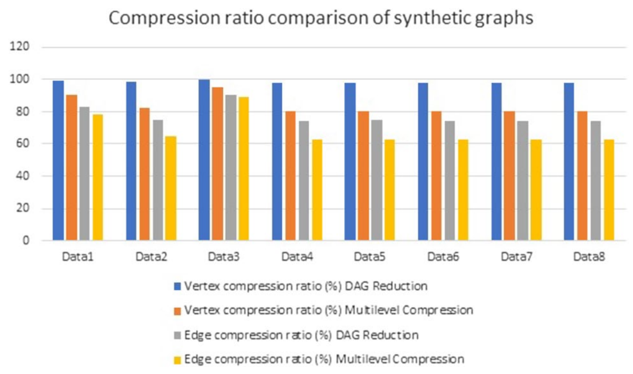


Fig. 6 Vertex and edge compression for G^ϵ and G_c

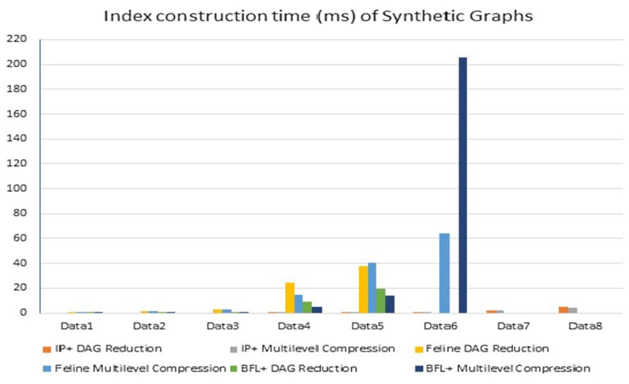


Fig. 7 Index construction time (ms) comparison for IP$^+$, Feline and BFL$^+$

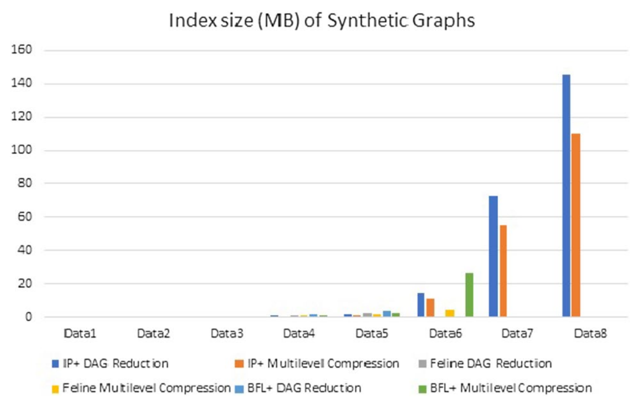


Fig. 8 Index size (MB) comparison for IP$^+$, Feline and BFL$^+$

6.3.4 Query Time

Table 9 shows the query time comparison where the query set contains 100,000 random queries for each of the dataset. We represented "–" in the table for which dataset the algorithm goes out of memory. From the table, only $IP+$ became able to process all of the datasets, whereas Feline and BFL$^+$ can process up to Data5 for G^ϵ and up to Data6 for G_c. We can see G_c shows significantly better performance than G^ϵ in Feline, whereas surprisingly only for Data4 G^ϵ is better than G_c. This could be happened because of graph structure and query set. The query time is much smaller in G_c than G^ϵ in all of the datasets for both IP^+ and BFL^+.

7 Discussion on *k*-Hop Reachability

Given two vertices v_1 and v_2 in a directed graph G and an integer k, the k-hop reachability asks whether v_1 can reach v_2 in k-hops, that is, whether there is a path from v_1 to v_2 that is of length k or less [5]. Due to the restriction on the length of the path, it is generally not beneficial to convert G into a DAG by merging the vertices in strongly connected components. Furthermore, an edge from v_1 to v_2 is no longer redundant even if there is a path from v_1 to v_2 of length 2 or more. Therefore, two vertices that share the same ancestors (resp. descendants) do not necessarily share the same parents (resp. children). Thus, it appears that neither the DAG reduction [22] nor the compression scheme in Sect. 5 can be applied to k-hop reachability.

However, in the real world, some graphs are naturally DAGs. For example, a paper citation network should have no cycles (for instance, two papers cannot cite each other). Moreover, linear chains and vertices that share the same parents and same children may naturally exist. Therefore, we can still use the compression scheme describe in Sect. 5 to compress the graph. In order to make the compression useful for k-hop queries, for each linear chain, we need to record the length of the chain as well as the position of the vertices in the chain. Given a k-hop reachability query, we can answer it using the a slightly modified algorithm from Algorithm 5.

1. If $LCA(u, v)$ is an independent set, then u cannot reach v, let alone reach v in k-hops.
2. If $LCA(u, v)$ is a linear module M, and N_1 is the child of $LCA(u, v)$ that contains u, and N_2 be the child of $LCA(u, v)$ that contains v, and $pos(N_1, M) < pos(N_2, M)$, then u can reach v, but we need to calculate the number of hops from u to v. To do that, we need to find all the linear modules on the path from the u to M in the decomposition tree, and find all the linear modules on the path from the v to M, and use the position number of the nodes u and v are in to compute the number of hops. For example, consider the decomposition tree in Fig. 5, and the two-hop reachability query from vertex 1 to vertex 7. We find the LCA of the two vertices to be LS3, which is a linear module. The two children of LS3 that contain vertices 1 and 7 are the leaf node 1 and IS1, and from the leaf noded 7 to LS3, there is a another linear module LS2, and the position of 7 in LS2 is 2. Therefore, we know the length of the chain from the first vertex to vertex 7 is 2. Since vertex 1 to IS1 have positions 0 and 1 in LS3, respectively, we know 1 can reach IS1 in one

Table 9 Query time (ms) for synthetic datasets

Dataset	IP$^+$		Feline		BFL$^+$	
	G^ϵ	G_c	G^ϵ	G_c	G^ϵ	G_c
Data1	23.35	**21.77**	286.04	**124.523**	25.03	**22.99**
Data2	31.71	**29.06**	1593.85	**914**	35.47	**34.22**
Data3	37.66	**34.02**	1988.83	**375.01**	41.61	**40.53**
Data4	66.17	**52.21**	**2069.84**	2484.5	58.26	**51.31**
Data5	76.42	**66.08**	58,884.8	**37,781.2**	82.11	**76.85**
Data6	79.52	**69.29**	–	**41,254.4**	–	**91.52**
Data7	82.3	**71.98**	–	–	–	–
Data8	89.02	**73.7**	–	–	–	–

hop; therefore, the path from vertex 1 to 7 has a length of 3. Hence, 1 cannot reach 7 in two hops. We may use straightforward pruning rules to prune paths longer than k rather than actually computing the length of the path in this step.

3. If $LCA(u, v)$ is the root of the decomposition tree, we can use a existing algorithm to find whether there is path from N_1 to N_2 of length no larger than k, and use a similar method to the above step to find whether there is a path from u to v of length $\leq k$.

It is obvious that k-reachability is more complicated, and the compression ratio may not be as big as for plain reachability queries. Whether the above approach is helpful for real graphs needs to be verified using experiments, and we leave it as our future work.

8 Conclusion

We presented a method to compress a DAG that has no redundant edges, using two types of modules, to obtain a decomposition tree. We showed how to use the decomposition tree to answer reachability queries over the original graph. Experiments show that for many real-world graphs and also for synthetic graphs, our method can compress the graph to much smaller graphs than DAG reduction, and reachability queries can be answered faster, and the index size can be smaller as well.

Acknowledgements This work is supported by Australian Research Council discovery Grant DP130103051. We thank the authors of [16, 18, 19, 21, 22] for kindly providing their code.

References

1. Agrawal R, Borgida A, Jagadish HV (1989) Efficient management of transitive relationships in large data and knowledge bases. In: SIGMOD, pp 253–262
2. Cai J, Poon CK (2010) Path-hop: efficiently indexing large graphs for reachability queries. In: CIKM, pp 119–128
3. Chen Y, Chen Y (2008) An efficient algorithm for answering graph reachability queries. In: ICDE, pp 893–902
4. Cheng J, Huang S, Wu H, Chen Z, Fu AW (2013) TF-label: a topological-folding labeling scheme for reachability querying in a large graph. In: SIGMOD
5. Cheng J, Shang Z, Cheng H, Wang H, Yu JX (2014) Efficient processing of k-hop reachability queries. VLDB J 23(2):227–252
6. Cohen E, Halperin E, Kaplan H, Zwick U (2002) Reachability and distance queries via 2-hop labels. SIAM, Philadelphia, pp 937–946
7. Fan W, Li J, Wang X, Wu Y (2012) Query preserving graph compression. In: SIGMOD, pp 157–168
8. Jagadish HV (1990) A compression technique to materialize transitive closure. TODS 15:558–598
9. Jin R, Ruan N, Dey S, Yu JX (2012) SCARAB: scaling reachability computation on large graphs. In: SIGMOD, pp 169–180
10. Jin R, Ruan N, Xiang Y, Wang H (2011) Path-tree: an efficient reachability indexing scheme for large directed graphs. TODS 36:7
11. Jin R, Wang G (2013) Simple, fast, and scalable reachability oracle. PVLDB 6:1978–1989
12. Jin R, Xiang Y, Ruan N, Fuhry D (2009) 3-hop: a high-compression indexing scheme for reachability query. In: SIGMOD, pp 813–826
13. Jin R, Xiang Y, Ruan N, Wang H (2008) Efficiently answering reachability queries on very large directed graphs. In: SIGMOD, pp 595–608
14. McConnell RM, de Montgolfier F (2005) Linear-time modular decomposition of directed graphs. Discrete Appl Math 145(2):198–209
15. Seufert S, Anand A, Bedathur SJ, Weikum G (2013) Ferrari: flexible and efficient reachability range assignment for graph indexing. In: ICDE, pp 1009–1020
16. Su J, Zhu Q, Wei H, Yu JX (2017) Reachability querying: can it be even faster? TKDE 29:683–697
17. Triβl S, Leser U (2007) Fast and practical indexing and querying of very large graphs. In: SIGMOD, pp 845–856
18. Veloso RR, Cerf L, Junior WM, Zaki MJ (2014) Reachability queries in very large graphs: a fast refined online search approach. In: EDBT, pp 511–522
19. Wei H, Yu JX, Lu C, Jin R (2014) Reachability querying: an independent permutation labeling approach. PVLDB, pp 1191–1202
20. Yano Y, Akiba T, Iwata Y, Yoshida Y (2013) Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In: CIKM
21. Yildirim H, Chaoji V, Zaki MJ (2012) GRAIL: a scalable index for reachability queries in very large graphs. VLDBJ 21:509–534
22. Zhou J, Zhou S, Yu JX, Wei H, Chen Z, Tang X (2017) DAG reduction: fast answering reachability queries. In: SIGMOD, pp 375–390