Reachability Queries in Very Large Graphs: A Fast Refined Online Search Approach

René R. Veloso[1], Loïc Cerf[2], Wagner Meira Jr[2], Mohammed J. Zaki[3,4]
[1]Universidade Est. Montes Claros, Brazil
[2]Universidade Federal de Minas Gerais, Belo Horizonte MG, Brazil
[3]Qatar Computing Research Institute, Doha, Qatar
[4]Rensselaer Polytechnic Institute, Troy NY, USA
rene.veloso@unimontes.br, {lcerf,meira}@dcc.ufmg.br, zaki@cs.rpi.edu

ABSTRACT

A key problem in many graph-based applications is the need to know, given a directed graph G and two vertices $u, v \in G$, whether there is a path between u and v, i. e., if u reaches v. This problem is particularly challenging in the case of very large real-world graphs. A common approach is the pre-processing of the graphs, in order to produce an efficient index structure, which allows fast access to the reachability information of the vertices. However, the majority of existing methods can not handle very large graphs. We propose, in this paper, a novel indexing method called FELINE (*Fast rEfined onLINE search*), which is inspired by *Dominance Graph Drawing*. FELINE creates an index from the graph representation in a two-dimensional plane, which provides reachability information in constant time for a significant portion of queries. Experiments demonstrate the efficiency of FELINE compared to state-of-the-art approaches.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph labeling, graph algorithms, path and circuit problems*

General Terms

Algorithms, Performance

Keywords

Reachability queries, online search, graph indexing

1. INTRODUCTION

Developing scalable methods for the analysis of large sets of graphs, including graphs that model biological and chemical structures is a challenging task. Further, the continuous growth of the size of the graphs, mainly in the web context, where they can reach tens of millions of nodes, makes such a task even harder and demands the development of efficient methods to handle these graphs.

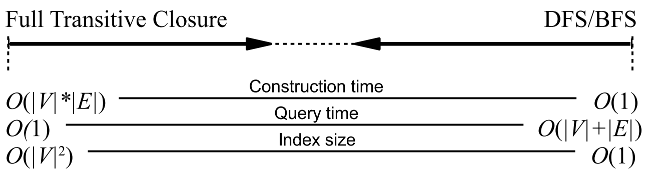


Figure 1: Reachability query tradeoffs

A common problem in several graph-based applications is to verify whether a vertex is reachable from another. Let $G = (V, E)$ be a directed graph, with V being its set of vertices and $E \subseteq V^2$ its set of edges. A reachability query $r(u, v)$ asks whether a vertex $v \in V$ is reachable from a vertex $u \in V$, i. e., whether there is a path from u to v in G (see Definition 1). This decision problem is often found in social networks analysis, where, for instance, it is necessary to learn whether there is a relationship between two entities, for security reasons, to provide conditional access to shared resources or in business intelligence [2, 13]. Reachability is also useful to determine the visibility between websites for link structure analysis [25].

Definition 1 (Reachability).

$$\forall (u, v) \in V^2,\ v \textit{ is reachable from } u, \textit{ denoted } r(u, v), \\ \textit{if and only if} \begin{cases} u = v \\ \textit{or} \\ \exists (u, w) \in E \wedge r(w, v) \end{cases} .$$

Definition 1 shows the reachability relation r. This reachability relation trivially is reflexive and transitive. It is not antisymmetric, hence not a partial order, because G may contain cycles. That is why the approaches first turn G acyclic, i. e., they build the graph $G' = (V', E')$ by condensation (folding every strongly connected component of G into one single vertex in V' and retaining in E' the edges between those components). Tarjan's algorithm [30] is used to identify the strongly connected components, i. e., the function $scc : V \to V'$. Its time complexity is $O(|V| + |E|)$. By using the acyclic graph G', the answer to the reachability query from $u \in V$ to $v \in V$ (in G) is the same as the answer to the reachability query from $scc(u) \in V'$ to $scc(v) \in V'$

(in G'). However the reachability relation in G' now is a partial order.

We follow the convention of existing approaches and assume that all input graphs have been transformed into their corresponding DAGs, i. e., we will refer the graph G' as the DAG G, omitting the application of function $scc(u)$ to every vertex $u \in V$. Thus, we will discuss methods for reachability only on DAGs.

As explained by Yildirim *et al.* [35], there are two basic approaches to answer reachability queries on DAGs, which are in two extremes of a spectrum (see Figure 1 extracted from [35]). The first approach (left side) is to pre-compute and store the full transitive closure of edges, which allows constant time queries, but requires a quadratic space complexity, making it infeasible to maintain the index in the case of very large graphs. The second approach (right side) is to employ a DFS or BFS search to verify the reachability, starting from vertex u to vertex v. This approach requires $O(|V| + |E|)$ time for each query, which is often unacceptable. There are some alternate approaches between these two basic ones, including ours, which aim to obtain a small index that is able to answer most queries in $O(1)$ time.

In a recent work, Jin *et al.* [18] divide the existing alternate approaches into three classes: *Transitive Closure Compression*, *Hop Labeling* and *Refined Online Search*. However, the same work states that the manipulation of large real-world graphs is not supported by the majority of the methods. In this paper, we demonstrate that it is possible to create a new and efficient reachability index for very large graphs, that is simple to understand and easy to implement. This new index is inspired by *Weak Dominance Drawing* and exposes a nice connection to graph drawing literature [3, 4, 14, 23].

Our reachability index represents non-planar DAGs in a two dimensional plane by assigning to each vertex u a unique coordinate (i. e., a pair of integers (x_u, y_u)) in the $\mathbb{N}^2$ plane, preserving the reachability relations between most pairs of vertices. Graphically, we translate reachability to the relation "being at the upper-right", that is, both coordinates of a vertex v, reachable from another vertex u, need to be greater than or equal to the respective coordinates of u, necessarily. When this geometrical relation holds, we verify the reachability through a search in a reduced part of the graph, identified from the index. When it does not, the non-reachability is decided in constant time.

Based on this index, we present in this paper a novel indexing method called FELINE (*Fast rEfined onLINE search*). In particular, we make the following original contributions:

- To our knowledge, FELINE is the first online search approach that brings the concepts of *Weak Dominance Drawing* applied to indexing very large graphs;

- When the non-reachability is not decided in $O(1)$, our method allows an effective pruning of the search space;

- Our experiments also show that FELINE generates a small index structure and outperforms the state of the art approach by more than 40% w.r.t. query and construction times;

- We show that FELINE can be improved by an existing boosting framework, called SCARAB [18].

2. RELATED WORK

As mentioned, Jin *et al.* [18] divides the existing approaches into three classes: *Transitive Closure Compression*, *Hop Labeling* and *Refined Online Search*. The methods of the first class [1, 8, 21, 26, 33, 34] compress the transitive closure of edges to assign to each vertex u a reachability compressed set. Thus, we determine the reachability between two vertices by verifying whether the destination vertex is in the reachability compressed set of the origin vertex. The second class is called *Hop Labeling* [10, 11, 16, 20, 27] and its methods use intermediate vertices to encode the reachability between two other vertices. In general, each node maintains a list of intermediate vertices that it can reach, and a list of intermediate vertices that can reach it. Consequently, to answer the queries, a *join* operation is made between the lists to determine the occurrence of vertices in common. In the third class, as proposed by [7, 28, 31, 35], the approaches are based on the use of online searches. In this case, they use a scheme for labeling the vertices in order to prune aggressively the search space, minimizing the number of vertices to be expanded. However, the original graph must be maintained in the memory to enable the searches, which is usually in DFS fashion. Thus, these approaches have the advantage of not requiring prior computation of the transitive closure in the construction of the index, and have a more intuitive index construction step, which allows its application to very large graphs, and the query costs, in the worst case, grow proportionally to the size of the graph.

Despite all the existing approaches, Yildirim *et al.* [35] showed in their work that very large graphs are not supported by the vast majority of them. For example, the recent PATHTREE [19], that had its scalability contested [18, 28, 35], since it performed well just on small graphs. Indeed, we found in the literature only three methods that can handle graphs with more than 100,000 edges, the Nuutila's INTERVAL [26, 33], GRAIL [35], FERRARI [28] and TF-Label [9].

Nuutila's INTERVAL [26, 33] extracts the complete transitive closure of the graph, and uses some lists of interval representation to compress any contiguous vertex segment. For instance, if the transitive closure of vertex u is the set of vertices identified by the integers $1, 2, 3, 4, 6, 7, 8, 9, 11, 12$, it can be compressed into three intervals: $[1, 4]$,$[6, 9]$ and $[11, 12]$. A bit-vector compression method, called PWAH (Partitioned Word Aligned Hybrid compression scheme), is used to compress these lists of intervals, allowing efficient operation with the compressed lists. Given the number of intervals' lists I, it constructs an index of size $O(I)$ in $O(|V| * |E|)$ time, and requires an $O(\log_2 I)$ time for each query

GRAIL (*Graph Reachability indexing via rAndomized Interval Labeling*) is an online search approach, where the index is formed by multiple intervals obtained by the traditional *min-post* strategy. *Min-post* strategy creates a unique interval $L_u = [s_u, e_u]$ for each vertex u, where s_u and e_u denote the start and the end of the interval. The ending value e_u is defined as $e_u = post(u)$, that is the post-order value of the vertex u, and the starting value s_u is defined as $s_u = min\{s_x | x \in children(u)\}$ if u is not a leaf and $s_u = e_u$, if u is a leaf. GRAIL employs some optimizations to speed up the query times, like *positive-cut filter* and *level filter* (these optimizations are also used in Feline and are explained later in this paper). Assuming the use of d intervals, GRAIL constructs an index of size $O(d|V|)$, in a

$O(d(|V| + |E|))$ time. Its query time ranges from $O(d)$ to $O(|V| + |E|)$.

Very similar to GRAIL, FERRARI (for Flexible and Efficient Reachability Range Assignment for gRaph indexind) [28] also uses intervals obtained from multiple post-order traversals. However, FERRARI employs a selective interval set compression, where a subset of adjacent intervals is merged into a small number of approximate intervals. The merge operation of the interval sets needs to be applied to each edge, so that every vertex will receive at most $deg(v) \times d$ intervals (for every vertex v and d intervals). Then, the time complexity for assigning the intervals is $O(\sum_v deg(v) \times d^2) = O(|E| \times d^2)$. Given d intervals of a vertex, FERRARI checks the target's intervals set using binary search since the set is in sorted order. The most relevant difference from GRAIL is that FERRARI uses a topological ordering of the vertices to limit the search space, pruning the search whenever all vertices to be searched have a topological ordering rank greater than the queried vertex. Its index size, construction time and query time are, respectively, $O(d|V|)$, $O(|E| * d^2)$, and ranging from $O(\log d)$ to $O(|V| + |E|)$.

Another work [18] presents a unified framework for reachability computing, called SCARAB (*standing for SCAlable ReachABility*), which is an efficient boosting approach that allows speeding up the queries. SCARAB proposes the use of a reduced graph extracted from the original graph, containing the so called “reachability backbone” vertex set. This reduced graph contains the main “access routes” between all vertices of the graph. To determine whether there exists a path between two vertices, SCARAB checks whether these vertices are connected to the backbone. SCARAB uses GRAIL as base method for searching and a search on the backbone determines whether they are reachable from each other. We understand that SCARAB is beyond the scope of our work, since it is complementary. However, preliminary experiments showed that SCARAB can be used to improve the Feline performance, as shown in Section 4.4.

A recent work [9] proposes TF-Label, an efficient and scalable indexing scheme, as its authors words. TF-label (where TF means Topological Folding) recursively folds the graph to reduce the final index and applies labels to every vertex, similarly to traditional *Hop Labeling* approaches. The approach assigns the sets $labels_{out}(u)$ and $labels_{in}(u)$ to each vertex u of the input graph, obtained from the topological folding step, then for a given query $r(u, v)$, it just needs to verify the intersection between $labels_{out}(u)$ and $labels_{in}(v)$ to answer the reachability. If a common element is found, u reaches v. Its index size is bounded by the number of labels sets, and its query time complexity is bounded by the intersection operation. The construction time is approximately linear with the in/out-degree of each vertex.

3. THE FELINE METHOD

The fundamental idea behind FELINE is to associate every vertex in V with a unique ordered pair of natural integers so that a partial order $\preceq$ over $\mathbb{N}^2$ is a superset of the partial order r over V. In other terms, given the index $i : V \to \mathbb{N}^2$, built by FELINE, and two vertices $(u, v) \in V^2$, the implication $r(u, v) \Rightarrow i(u) \preceq i(v)$ always holds. In this way, if $i(u) \npreceq i(v)$ then no traversal of the graph is needed to negatively answer the reachability query from u to v. We will also see that, whenever $i(u) \preceq i(v)$, the index allows to discard vertices in V, i. e., to reduce the search of a path in G to the search of a path in a sub-graph of G. Here is the definition of the order $\preceq$ over $\mathbb{N}^2$:

Definition 2 (Partial order $\preceq$). $\forall(x_u, y_u, x_v, y_v) \in \mathbb{N}^4$, $(x_u, y_u) \preceq (x_v, y_v) \Leftrightarrow x_u \leq x_v \wedge y_u \leq y_v$.

The reflexivity, antisymmetry and transitivity of $\preceq$ directly derive from those of $\leq$. Geometrically, $(x_u, y_u) \preceq (x_v, y_v)$ means that (x_v, y_v) is in the upper-right quadrant of the two-dimensional Cartesian system with (x_u, y_u) as the origin and we say u dominates v. Notice that testing whether this relation holds is performed in constant time.

The (x, y) coordinates attributed to every vertex of G are called *Dominance Drawing* of G [23]. The method is inspired on a *weak dominance drawing* [23], which guarantees that $\forall(u, w) \in E, i(u) \preceq i(w)$. From Def. 1, we therefore have:

Lemma 1. $\forall(u, v) \in V^2$,

$$r(u,v) \Rightarrow \begin{cases} u = v \\ or \\ \exists w \in V \setminus \{u\} | i(u) \preceq i(w) \wedge r(w, v) \end{cases}.$$

The recursive application of this lemma and the transitivity of $\preceq$ together prove the following theorem:

Theorem 1. $\forall(u, v) \in V^2$, $r(u, v) \Rightarrow i(u) \preceq i(v)$.

With this theorem in hand, we can complement Def. 1:

Definition 3 (Alg. definition of reachability).

$$\forall(u,v) \in V^2, r(u,v) \Leftrightarrow \begin{cases} u = v \\ or \\ \exists(u, w) \in E | i(w) \preceq i(v) \wedge r(w, v) \end{cases}.$$

From a logical perspective, writing “$i(w) \preceq i(v) \wedge r(w, v)$” instead of “$r(w, v)$” does not bring anything since $r(w, v) \Rightarrow i(w) \preceq i(v)$ (by Theorem 1). Nevertheless, it is algorithmically interesting. It means that the recursive search for a path from u to v (in G and starting from u) can be aborted whenever an edge leads to a vertex w such that $i(w) \npreceq i(v)$. As a consequence, only a sub-graph of G is traversed.

With the help of FELINE's index, answering the reachability query from $u \in V$ to $v \in V$ becomes an efficient two-step process:

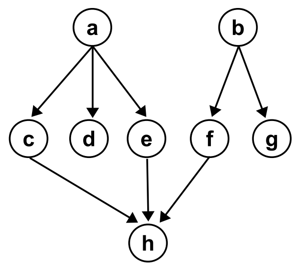
1. Test whether $i(u) \preceq i(v)$ and answer negatively if $i(u) \npreceq i(v)$ (by Theorem 1);
2. Otherwise search in G a path from u to v that only passes by vertices whose indexes are $\preceq i(v)$.

3.1 Graphical Meaning

Given a DAG G, every vertex is associated with a pair $(x, y) \in \mathbb{N}^2$ so that, for any two vertices $u, v \in G$, if there is a directed path from u to v, then the x-coordinate of u is less than or equal to the x-coordinate of v and y-coordinate of u is less than or equal to the y-coordinate of v. Figure 2 shows an example of a small DAG and the index of size $O(|V|)$.

The resulting index can be represented graphically. The pair of integers (x, y) associated with a vertex is understood as coordinates in the Cartesian plane.

Theorem 1 graphically means that, for a given vertex u, we have only to check its upper-right quadrant, in order to



	X	Y
a	1	4
b	5	1
c	2	7
d	3	6
e	4	5
f	6	3
g	8	2
h	7	8

(A) (B)

Figure 2: The drawing of a small DAG: in (A) the DAG; in (B) the related index, where each row represents the coordinate of a vertex.

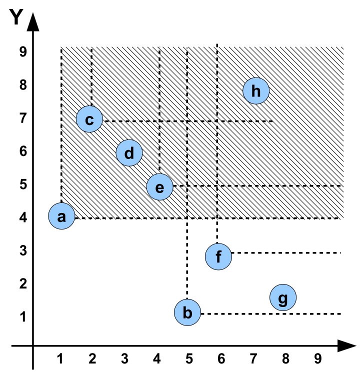


Figure 3: Example of dominance regions.

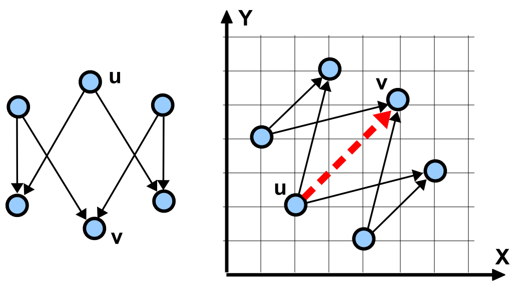


Figure 4: Example of exception between u and v. The dashed arrow is a *falsely implied path* or a *false-positive*.

identify the vertices that can be reachable from u. If not, the reachability is negatively answered in constant time.

The $\preceq$ relations between the vertices (or points in the plane) are illustrated in Figure 3, where the dashed lines express the reachability area starting from each vertex. For example, consider the vertices a and h. For $r(a, h)$, we necessarily have $i(a) \preceq i(h)$ (by Th. 1) and, indeed, the hatched area starting from point $i(a) = (1, 4)$ includes the point $i(h) = (7, 8)$. We can also see in the figure that d is not in the upper-right quadrant of b, i. e., $i(b) \npreceq i(d)$. Using the contraposition of Th. 1, we conclude that d is not reachable from b.

Unfortunately, the index does not allow to positively answer a query in constant time. In other terms, Th. 1 states that $\forall(u, v) \in V^2$, $r(u, v) \Rightarrow i(u) \preceq i(v)$, but the reverse implication does not always hold. Figure 4 shows a crown DAG known as S_3^0 graph. In the related index, we have $i(u) \preceq i(v)$ even if $r(u, v)$ is false, represented by a dashed arrow between the vertices u and v (*false-positive*). Another example is the vertex d in Figures 2 and 3, where, for the vertex h, $i(h)$ is in the upper-right quadrant of $i(d)$, but it is not reachable according to the graph (from Figure 2-(A)).

It is important to notice that some graphs, such as S_3^0, do not admit a $2D$ index which is free of false-positives. In fact, such problematic graphs exist for the construction of any nD index with n arbitrarily large [14, 23]. The problem of minimizing the number of false-positives is known to be NP-hard [23, 24], but we can generate an approximate locally optimal solution that minimizes the number of false-positives. (Another solution is shown later in this paper by applying an optimization called *positive-cut filter*.)

3.2 Index Construction

Algorithm 1 generates the coordinates that will compose the FELINE index. The x coordinates are first determined by a topological ordering algorithm (line 2), resulting in the set of coordinates X. Any topological ordering algorithm can be used (an $O(|V| + |E|)$ time is found in [12]).

Algorithm 1: Index construction

```
Data: $(V, E)$, a directed acyclic graph whose set of vertices, $V$,
      is the set of integers from 1 to $|V|$ and $E \subset V^2$
Result: $(X, Y)$, two vectors of size $|V|$ such that $\forall u \in V$,
        $i(u) = (X_u, Y_u)$
1  begin
2      $X \leftarrow$ TopologicalOrdering$(V, E)$;
3      $Y \leftarrow \emptyset$;
4      heads $\leftarrow (\emptyset, \ldots, \emptyset)$;
5      $d \leftarrow (0, \ldots, 0)$;
6      forall the $(u, v) \in E$ do
7          heads$_u \leftarrow$ heads$_u \cup \{v\}$;
8          $d_v \leftarrow d_v + 1$;
9      roots $\leftarrow \{v \in V \mid d_v = 0\}$;
10     while roots $\neq \emptyset$ do
11         $u \leftarrow \arg\max_{v \in \text{roots}}(X_v)$;
           // Append u to Y
12         $Y \leftarrow (Y, u)$;
           // Update d and the roots set
13         roots $\leftarrow$ roots $\setminus\{u\}$;
14         forall the $v \in$ heads$_u$ do
15             $d_v \leftarrow d_v - 1$;
16             if $d_v = 0$ then
17                 roots $\leftarrow$ roots $\cup\{v\}$;
```

To compute the Y set with y coordinates, we use a heuristic (lines 3 to 17) proposed by Kornaropoulos in [24]. Each step of the heuristic makes a decision about optimal locality of the final position of the vertices minimizing the number of false-positives. It performs repeated deletions of vertices with no predecessors (i. e., root nodes) to generate a new topological ordering Y. Given an initial set of roots (line 9), the algorithm iteratively chooses the root with the highest rank in X (line 11), a particular case of Kahn's algorithm [22].

The arg max operator (line 11) tries to avoid an exception between the chosen vertex and the vertices of previous iterations, by choosing the root vertex with the higher rank value. This choice was proved to be locally optimal in [24], since it selects a root vertex u that generates zero falsely implied paths to any vertex in *roots*. After a root is selected and removed from the set, new vertices with no predecessors may appear and the set needs to be updated (lines 13 to 17).

For instance, consider the DAG of Figure 2-(A). A DFS-based topological ordering can generate the set X with the vertices $\{a, c, d, e, b, f, h, g\}$ associated with x coordinates with rank values from 1 to 8. The algorithm first fills the *roots* set with $\{a, b\}$ (line 9). Then, in line 11, it chooses the root vertex b (with the rank value 5, in X) and puts it into the Y set (in the first position). Next, in line 13, b is removed from *roots*, which is updated with the new roots f and g, and now $roots = \{a, f, g\}$. Then, as g has the higher rank in X, it is also removed from *roots* and inserted into the second position of Y. Notice that g has no descendants, and, at that moment, $Y = \{b, g\}$ and $roots = \{a, f\}$. The vertex f is the next chosen and $Y = \{b, g, f\}$. In Figure 3, the vertices b, g, and f have the y coordinates in 1, 2 and 3, respectively. The algorithm continues until the Y set is completed.

3.3 Reachability Queries

Algorithm 2 is a pseudo-code that summarizes FELINE's query strategy. As mentioned earlier, for two vertices u and v, it first checks whether u is equal to v, because of the reflexive property or in case the search has reached the destination. In line 4, if $i(u) \npreceq i(v)$ is true, we can immediately stop the search, by contraposition of Th. 1. This last step is known as *negative-cut*, because we need no search to conclude about the non-reachability between u and v. On the other hand, if $i(u) \preceq i(v)$, no conclusion can be reached immediately. In this case, FELINE needs to explore all vertices inside the region between u and v recursively (in DFS).

Algorithm 2: Reachable

```
    Data: $(u, v) \in V^2$, two vertices of the DAG
    Result: $r(u, v)$, whether $v$ is reachable from $u$
1   begin
2       if $u = v$ then
3           return true;
4       if $i(u) \preceq i(v)$ then
5           forall the $w \in heads_u$ do
6               if $Reachable(w, v)$ then
7                   return true;
8       return false;
```

Let $\tau : V \to \{1, 2, ..., n\}$ denote a topological ordering of the vertices, for all $(u, v) \in E$ it holds $\tau(u) < \tau(v)$. The sets X and Y store the topological ordering ranks computed in Algorithm 1, then, for all $(u, v) \in E$, it is true that $X_u < X_v$ and $Y_u < Y_v$.

3.3.1 Discussion on FELINE's performance

Regarding online search approaches, in the case of positive queries [1] (or even false-positives), the query time may increase as new vertices are expanded by DFS. However, due to the two topological orderings, the FELINE pruning method can discard those expansions that do not reach the searched vertex. Figure 5 illustrates the search associated with a positive query $r(u, v)$ in GRAIL, FERRARI and FELINE, more specifically, the triangle represents the search space composed by the vertices that are included in u's interval. For a better understanding, Figure 6 shows an example.

[1]we use the term *positive query* referring to those queries that receive a positive answer about the reachability, as in opposite, we use the term *negative query*

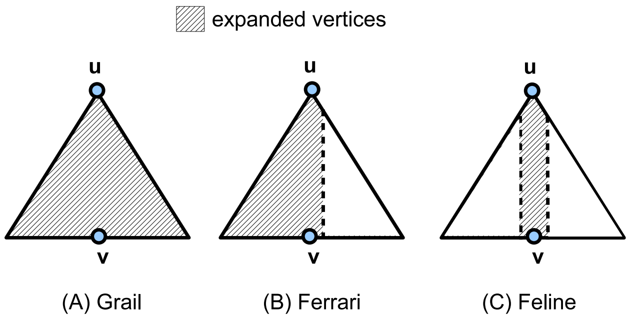


Figure 5: The search space of the three online search approaches: GRAIL, FERRARI and FELINE. The triangles represent the vertices expanded by a DFS. The hatched areas are the branches not pruned by the topological orderings.

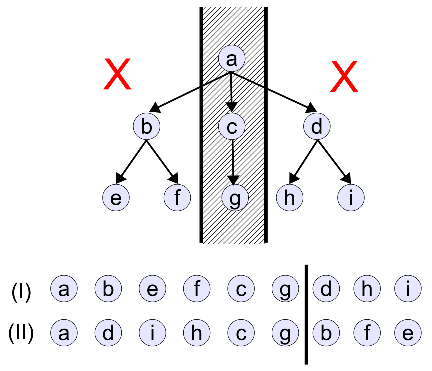


Figure 6: This figure shows a didactic example of a DAG and its two topological orderings obtained by Algorithm 1. Given a query $r(a, g)$, all vertices after g in (I) and (II) are discarded. This strategy reduces the search space to only vertices a, c and g.

In all methods, the DFS will proceed expanding vertices until no more vertices can be expand. But, specially in the case of false-positives, the goal vertex v will never be found by GRAIL and all vertices will be always expanded (or until the level of v is reached). FERRARI avoids this case by pruning vertices with rank higher than v in the topological ordering, limiting the search to only a part of the search space.

Figure 7 depicts the equivalence between the search space explored by both GRAIL and FERRARI to the search space explored by FELINE. It shows the pruning behaviour of FELINE when it applies the same search concept used by

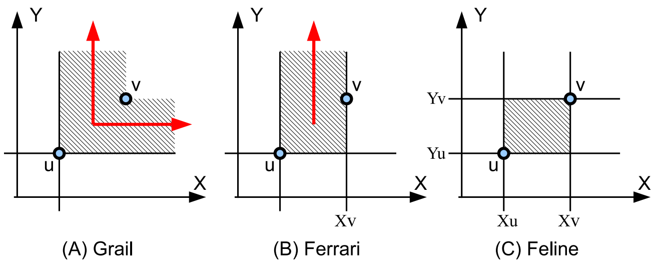


Figure 7: Pruning behaviour of FELINE when employing the same search strategies of GRAIL and FERRARI. The red arrow indicates the possibility of branches to be explored by DFS.

GRAIL and FERRARI. As FERRARI considers only one topological rank of the goal vertex, it works as FELINE's pruning in just one dimension (e. g., on x-axis, in the figure). Although GRAIL employs a directed search, it may explore paths that will never reach v, being equivalent to FELINE without bounds checking in each dimension.

3.3.2 *Computational Complexity*

It is easy to see that Algorithm 2 takes time $O(1)$ when, for two vertices $u, v \in G$, either u and v belong to the same strongly-connected component or the weak dominance relation does not hold (i. e., $i(u) \not\preceq i(v)$). However, when $i(u) \preceq i(v)$, the reachability is decided by a DFS search in a generally small part of the graph G identified by the index. The worst case is to traverse the whole graph G, i. e., $O(|V|+|E|)$. However, in practice, the worst case will rarely happens. The whole graph will be traversed only for those graphs with just one root and, for queries starting from the root vertex, the DFS needs to choose the wrong paths to the target vertex. Furthermore, an optimization can prune the search before its termination.

Regarding the time for index construction, the complexity of Algorithm 1 is $O(|V|log|V| + |E|)$, because all edges are enumerated (once in line 6, and again in line 14) and *roots* is stored as a max-heap structure, where every vertex is inserted once ($O(log|V|)$), so that line 11 has an $O(1)$ cost. X, Y and d are simple arrays and *heads* is an array of arrays. Since the algorithm uses a $O(|V|+|E|)$ topological ordering, its final complexity is still $O(|V|log|V| + |E|)$.

3.4 Optimizations

3.4.1 *Positive-Cut Filter*

Several approaches described in the literature use *spanning-trees* in the index construction [1, 7, 31, 34, 35]. According to Yildirim *et al.* [35], in a tree, indexing using a traditional *min-post* algorithm is enough to answer some queries in constant time. This strategy is called *positive-cut filter.*

As mentioned in Section 2, the *min-post* strategy creates a unique interval $I_u = [s_u, e_u]$ for each vertex u, where s_u and e_u denotes the start and the end of this interval. For those paths restricted to tree edges, reachability can be guaranteed (only positive queries). Thus, if $I_v \subseteq I_u$, we can conclude that u reaches v. However, we can state nothing about the non-tree edges, explaining why GRAIL uses this strategy as a filter, i.e., one more pruning step before searching in the graph. FERRARI also employs the same strategy.

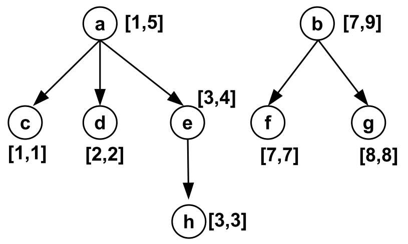


Figure 8: Min-post indexing for a spanning-tree extracted from the DAG of Fig.2. Each vertex is labeled with its *min-post* interval.

Figure 8 shows a spanning-tree (a spanning-forest, in this case) extracted from the DAG of Figure 2 and indexed with the *min-post* strategy (the labels represent the index). For instance, consider the vertex h of Figure 8. We can conclude that the query $r(a, h)$ will return true, without a search, because $[3, 3] \subseteq [1, 5]$. However, we can say nothing about the query $r(b, h)$, because there is no tree edges connecting the two vertices.

FELINE adds two extra steps in Algorithm 1 to compute the positive-cut intervals. First, it is the extraction of a spanning-tree (that may be performed by the topological ordering in line 2), and, second, the application of *min-post* strategy. To use the positive-cut filter, Algorithm 2 needs to be changed to Algorithm 3, where the new lines 1 and 2 verify the intervals assigned to each vertex by *min-post.*

3.4.2 *Level Filter*

Another filter used in FELINE (also in GRAIL and FERRARI) is the *Level Filter* [35], which is an extra indexing strategy that assigns to each vertex an integer number according to its level, representing a negative-cut.

The level of a vertex v (l_v) can be defined as its depth [5]: if v has no immediate predecessors, the $l_v = 0$; otherwise, $l_v = \max_{u:(u,v)\in E} l_u + 1$. According to Bender *et al.* [5], the level of a vertex induces its topological ordering: if u precedes v in the Dag G and $u \neq v$, then $l_u < l_v$. Algorithm 3 applies this filter (line 6).

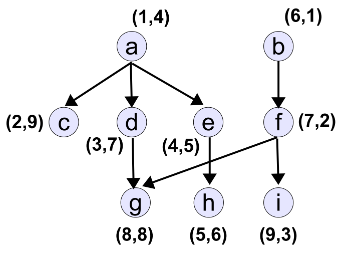


Figure 9: Example where the vertex h does not reach g, but there is a false-implied path between them (g is in domination area of h). However, g and h are in the same level, and the *level filter* prune the search. The numbers represents the coordinates attributed by FELINE.

Figure 9 helps to understand the applicability of level fil-

Graph	vertices	edges	Cluster-coeff	Eff-diameter	roots	leafs
Arxiv	6000	66707	0.35	5.48	961	624
Yago	6642	42392	0.24	6.57	5176	263
Go	6793	13361	0.07	10.92	64	3087
Pubmed	9000	40028	0.10	6.32	2609	4702
citeseer	10720	44258	0.28	8.36	4572	1868
Uniprot22m	1595444	1595442	0.00	3.3	1556157	1
Cit-patents	3774768	16518947	0.09	10.5	515785	1685423
citeseerx	6540401	15011260	0.06	8.4	567149	5740710
Go-uniprot	6967956	34770235	0.00	4.8	6945721	4
Uniprot100m	16087295	16087293	0.00	4.1	14598959	1
Uniprot150m	25037600	25037598	0.00	4.4	21650056	1

Table 1: Datasets

ter. The example shows that a false-positive query $r(h, g)$ could be pruned in constant time just by verifying that h and g are at the same level.

```
Algorithm 3: Reachable
    Data: $(u, v) \in V^2$, two vertices of the DAG
    Result: $r(u, v)$, whether $v$ is reachable from $u$
1   begin
2       if $I_v \subseteq I_u$ then
3           return true;
4       if $u = v$ then
5           return true;
6       if $i(u) \preceq i(v)$ and $l_u < l_v$ then
7           forall the $w \in heads_u$ do
8               if Reachable$(w, v)$ then
9                   return true;
10      return false;
```

4. EXPERIMENTS

4.1 Datasets

4.1.1 Benchmark: Real graphs

We used 5 small (i. e., less than 100,000 vertices) and dense real-world DAGs: *Arxiv*[2], *Citeseer*[3], *Go*[4], *Pubmed*[5] and *Yago*[6]. We also used 6 large real-world DAGs (sparse and dense): *Cit-Patents*[7], *Citeseerx*[8], *Uniprot22m*, *Go-uniprot*, *Uniprot100m* and *Uniprot150m*[9]. Table 1 shows some characteristics of the graphs, such as the number of vertices and edges, clustering coefficient, effective diameter (instead of diamenter) available from [35] (the authors used the SNAP software[10] to compute these values). The effective diameter (or effective eccentricity) is an estimated size of the path in which 90% of all pairs of vertices connected are reachable from each other [6]. The effective diameter is more robust than the diameter and has been successfully used to analyse topological properties of Internet graphs [6, 29].

[2]arxiv.org
[3]citeseer.ist.psu.edu
[4]www.geneontology.org
[5]www.pubmedcentral.nih.gov
[6]www.mpi-inf.mpg.de/suchanek/downloads/yago
[7]snap.stanford.edu/data/cit-Patents.html
[8]citeseerx.ist.psu.edu
[9]www.uniprot.org
[10]snap.stanford.edu/snap/

4.1.2 Synthetic graphs

We also performed experiments using synthetic random graphs (DAGs) generated to test the scalability of the approaches. Table 2 shows these datasets. Each graph was generated according to [35]. At first, given a predetermined set of vertices, we generated a permutation of this set, in practice a different topological ordering. Then, for a given number of edges, we randomly select two vertices and connect them, respecting their topological order. The size of those graphs make the computation of the clustering coefficients and effective diameters prohibitive.

Graph	vertices	edges
50M	50M	50M
60M	60M	60M
70M	70M	70M
80M	80M	80M
90M	90M	90M
100M	100M	100M
200M	200M	200M
50M-5	50M	250M
50M-10	50M	500M
100M-5	100M	500M
100M-10	100M	1000M

Table 2: Synthetic Dataset

4.2 Experimental methodology

GRAIL [35], FERRARI, Nuutila's INTERVAL [26, 33] and TF-Label [9] are considered the most efficient methods for answering reachability queries. Like FELINE, GRAIL and FERRARI also employ an online search where the graph should remain in memory along with the generated index. INTERVAL and TF-Label are based on the compression of the transitive closure, generating a self-sufficient index, that is, queries can be answered based only on the index, without the need to maintain the original graph in memory.

Fully optimized versions of the baselines were used in the experiments (following the authors' recommendations). All implementations (GRAIL, FERRARI, INTERVAL and TF-Label) were provided by their authors and are in C++, as well as FELINE. All FELINE's implementations use the *positive-cut filter* and the *level filter*.

The experiments with small graphs (i. e., with less than 100,000 edges) were performed on an Intel(R) Xeon(R) CPU E5620 (8-core, 2.40GHz) machine with 32G ram. For larger graphs, we used an Intel(R) Xeon(R) CPU E5620 (8-core, 2.40GHz) machine with 96G ram. However, all implementations are single threaded.

4.2.1 Queries

We generated a specific set of queries for each graph (real

	Construction Time					Query Time				
Graph	GRAIL	INTERVAL	FERRARI	TF-Label	FELINE	GRAIL	INTERVAL	FERRARI	TF-Label	FELINE
Arxiv	13.607	48.136	30.085	7885.118	**5.533**	745.230	**28.087**	166.103	164.076	493.092
Yago	11.714	13.707	18.501	50.998	**4.206**	27.810	16.099	33.119	11.689	**9.893**
Go	8.160	7.504	10.583	30.715	**3.333**	66.344	**18.193**	29.209	18.597	81.846
Pubmed	13.967	27.392	32.609	86.405	**5.528**	32.269	12.614	23.530	10.744	**9.997**
Citeseer	17.403	31.409	34.591	154.177	**7.356**	37.842	17.373	32.247	15.814	**13.226**
Uniprot22m	7121.763	1189.403	1083.154	2546.216	**818.732**	36.524	87.393	23.263	19.670	**18.115**
Cit-patents	22575.605	504209.060	28157.440	253828.562	**6065.227**	91.998	74.917	87.360	62.005	**41.291**
Citeseerx	25400.748	8049.076	18134.930	104418.257	**5519.161**	97.654	47.417	61.792	91.149	**40.292**
Go-uniprot	45998.303	23002.793	29491.370	70984.078	**6051.333**	37.633	112.861	213.040	30.319	**22.817**
Uniprot100m	93990.593	13279.480	14223.680	48741.968	**10533.610**	53.373	117.323	60.839	48.084	**26.507**
Uniprot150m	155546.666	21756.457	25738.860	70321.609	**17745.300**	63.384	121.837	61.466	55.397	**27.764**

Table 3: Construction and query times for real graphs. Average times (in milliseconds).

or synthetic) and we randomly selected 500k pairs of vertices for each set. We submit the respective set of queries to each dataset, and all results shown are the average values of 10 executions. With these sets of queries, we were able to estimate the performance in terms of query times, construction time of the indexes and their size.

All datasets, source code of the algorithms and test sets are available at `http://www.dcc.ufmg.br/˜ renerv/feline`.

4.3 Results

4.3.1 *Index construction time*

Table 3 shows the average values obtained from the construction times of indexes (in milliseconds). Best results are highlighted with a gray background. Figure 13 summarizes the times for synthetic graphs.

For small graphs, the construction time of FELINE is up to 3 times faster than GRAIL, FERRARI, INTERVAL and TF-Label. For the large graphs, the performance of the FELINE is up to 10 times better than the others.

We applied the Friedman test [15] to the results to obtain their statistical significance. At a confidence level of 0.1, FELINE's performance is significantly better in all cases. Proceeding to the Nemenyi post-hoc test [17], we plot the diagram of the Fig. 10, which represents the critical difference of performance between the approaches. In the diagram, the axis represents the average ranks of the methods, from 1 (best) to 4 (worst). When comparing all the methods against each other, we connect the groups that are not significantly different by a bold line. The critical difference (CD) is also shown and if the distance between the ranks of two methods is greater than the CD, then the methods are not grouped. For instance, the group formed by INTERVAL, GRAIL and FERRARI indicates that the differences of their performances are not statistically significant.

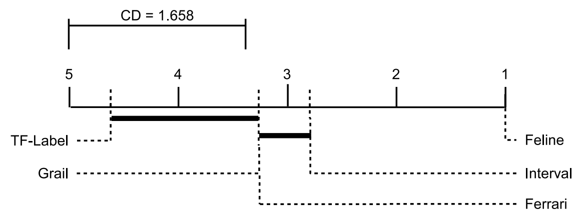


Figure 10: The critical difference diagram for the construction times.

4.3.2 *Query answering time*

The average times to answer a query are also shown in Table 3. The INTERVAL method achieved good results but, as we will show, could not build the indexes of the larger synthetic graphs. Again, at confidence level of 0.1, the Friedman test shows that the performances are significantly different. However, as detailed in Fig. 11, FELINE is competitive with INTERVAL and TF-Label, since they are in the same group. However, as FELINE is faster than GRAIL and FERRARI, we can state that FELINE has the best query time (considering the real datasets). Indeed, FELINE is typically 2 times better.

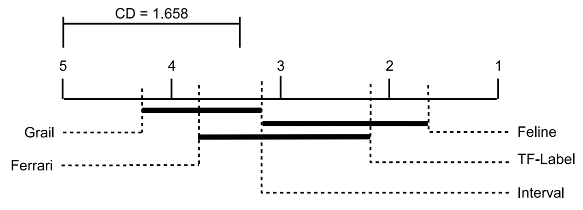


Figure 11: The critical difference diagram for the query times.

4.3.3 *FELINE-B approach*

In many graphs, the out-degrees and in-degree follow different distributions. To verify whether this difference can influence the FELINE's index, we plot the indexes for some graphs and its reversed version, i. e., for each DAG G, we generate another DAG $G^T = (V, E^T)$, where E^T is the set of edges with its reversed directions. We notice that the vertices of the normal and the reversed graphs present different placements. This occurs because reversing the edges of a DAG, due to the in-degree and out-degree distributions of its vertices, the number of successors (and predecessors) of each vertex changes, as well as the number of roots and leafs of the DAG. Figure 12 shows plottings from four DAGs indexes: Arvix, Yago, Go and Pubmed (Their sizes make possible the plottings). Obviously the query $r(u, v)$ in the DAG G is equivalent to the $r(v, u)$ in the DAG G^T, but, as the vertices have different coordinates in each index, each one performs different to the same query.

Considering the plottings, we decided to investigate whether the use of reversed graphs may contribute to improve the performance of FELINE. We then included new experiments where FELINE generates a *reversed index* (from the reversed graph). The version that used a reversed index is called FELINE-I. Table 4 summarizes the construction and

	Construction Time			Query Time		
Graph	FELINE	FELINE-I	FELINE-B	FELINE	FELINE-I	FELINE-B
Arxiv	5.533	5.467	10.760	493.092	894.211	**420.806**
Yago	4.206	3.776	8.090	9.893	24.754	**9.435**
Go	3.333	3.737	6.706	81.846	**34.753**	44.390
Pubmed	5.528	5.465	10.700	9.997	**9.006**	9.517
Citeseer	7.356	6.821	14.102	13.226	19.508	**11.608**
Uniprot22m	818.732	740.511	1567.305	**18.115**	18.964	19.461
Cit-patents	6065.227	5994.798	11913.330	41.291	**27.894**	32.226
Citeseerx	5519.161	6242.018	11243.650	**40.292**	56.193	41.777
Go-uniprot	6051.333	6616.405	12202.540	**22.817**	53.354	24.392
Uniprot100m	10533.610	9417.537	20373.920	**26.507**	28.410	29.837
Uniprot150m	17745.300	15985.690	34359.940	**27.764**	29.749	31.570

Table 4: Results for FELINE-B. Average times (in milliseconds).

query times for FELINE-I.

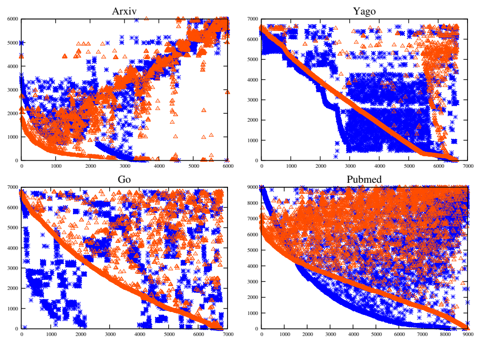


Figure 12: Indexes plotting. Legend: $*$ for normal and ▲ for reversed. The normal or reversed indexes can result in different performances.

The observed gains of FELINE-I, for some datasets, indicates that the reversed index can be used to compose a new and more efficient prune strategy. This new strategy is based on looking at both indexes (normal and reversed), i. e., in the intersection of the reachable areas (two weak dominance tests at line 4 of Algorithm 2). FELINE-B (bidirectional) implements this new strategy. For a query $r(u, v)$, all vertices out of the (u, v) area in the normal index and the (v, u) area in the reversed one, are discarded. The performance of FELINE-B is compared with FELINE and FELINE-I in Table 4. Notice that the average time for index building almost doubled, but FELINE-B results are close to the best single index for each dataset, and better than the average of both FELINE and FELINE-I.

4.3.4 *Synthetic Datasets*

The construction times and query times are reported in Figures 13 and 14, respectively. Notice that for the 200M, 50M-5, 50M-10, 100M-5 and 100M-10 datasets shown, there are no results for the INTERVAL and TF-Label, because during the experiments they failed with these datasets. FERRARI registered an execution time up to 8 hours for the 100M-10 dataset and was cancelled.

These results show that FELINE and FELINE-B are scalable and truly competitive with the state of the art approaches. With the experiments, we can demonstrate that FELINE has a very stable high performance in construction time and FELINE-B has the best query times.

4.3.5 *Index size*

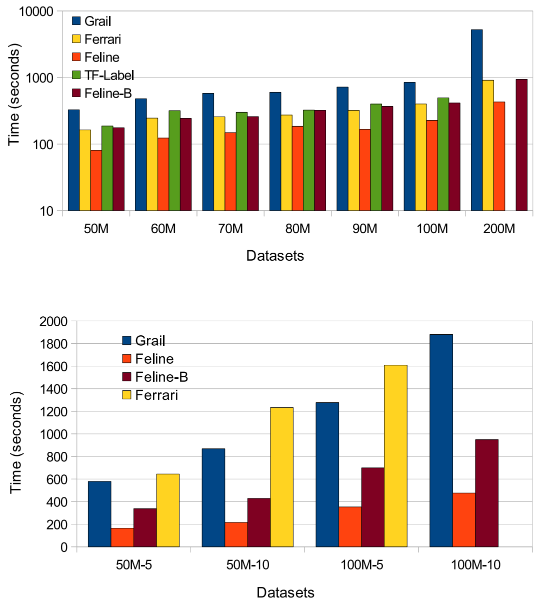


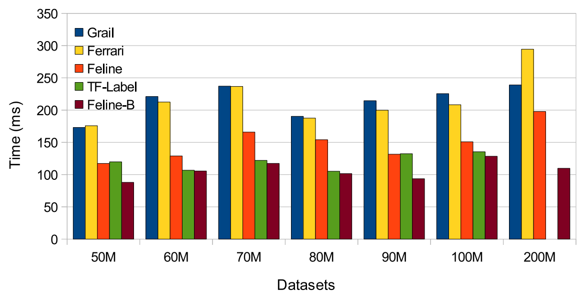
Figure 13: Construction times for synthetic graphs. At a confidence level of 0.1, all results are significantly different.

Figures 15 and 16 show the size of the indexes built by FELINE. The figures do not show the values for the FELINE-I, because its index size is the same of FELINE.

Due to its multiple intervals, GRAIL generates indexes that are larger than those generated by FELINE, which are 2 times greater when $d = 3$ and 4 times greater when $d = 5$. Since FELINE-B generates an index of size that is proportional to the normal and reversed graphs (with positive-cut and level filters), its index is greater than the FELINE's index (FELINE and FELINE-I), but it is not exactly twice as big because each filter is applied just once in FELINE-B, only on the normal index.

4.4 Scarab Framework

According to [18], the SCARAB framework (see Section 2) can speed up the reachability queries. Although it is not the aim of this paper to investigate the applicability of boosting methods, we use the framework to show that FELINE can also take advantage of it. For this, we implemented a



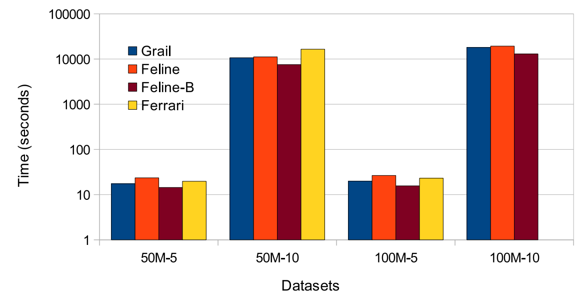
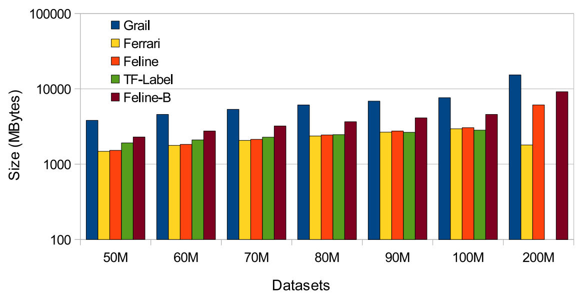


Figure 14: Query times for synthetic graphs. At a confidence level of 0.1, all results are significantly different.



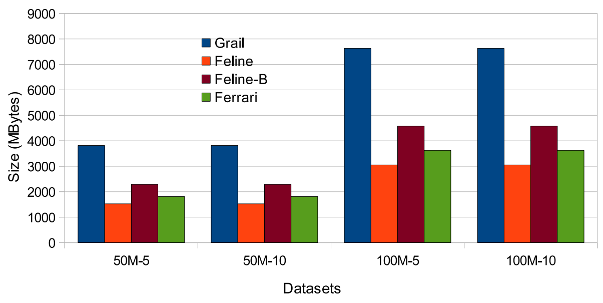


Figure 16: Index sizes for synthetic graphs. At a confidence level of 0.1, all results are significantly different.

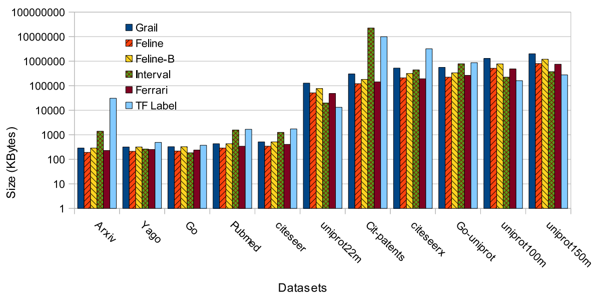


Figure 15: Index sizes for real graphs.

Graph	FELINE-SCAR	GRAIL-SCAR
Arxiv	**203.761**	319.452
Yago	**8.691**	20.760
Pubmed	**8.703**	17.932
citeseer	**10.539**	19.595
uniprot22m	**23.033**	63.202
Cit-patents	**30.749**	52.908
citeseerx	**35.248**	171.752
Go-uniprot	**28.025**	74.649
uniprot100m	**30.970**	85.538
uniprot150m	**33.068**	91.111

Table 5: Query times for SCARAB implementations. At a confidence level of 0.1, all results are significantly different.

version called FELINE-SCAR, derived from the FELINE-B method. In theory, any existing approach can be boosted with SCARAB, but we were not able to do so for INTERVAL and TF-label due to the complexity of the framework to handle "offline" approaches, since is necessary some modification on the original approaches. GRAIL-SCAR were provided by the authors and it was acclaimed as the best method in [18].

Table 5 shows the results of some experiments performed in the same conditions applied before (500k queries, 10 executions and the same machines), and following the recommendations of SCARAB's authors. Although some gains were observed with the SCARAB versions for some datasets, at a confidence level of 0.1, FELINE-SCAR and GRAIL-SCAR are significantly different, as presented in Figure 17.

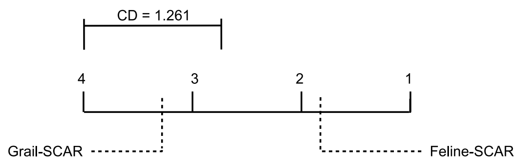


Figure 17: The critical difference diagram for Scarab experiments.

4.5 Discussion

For all datasets we can notice that, despite the particular characteristics of the graphs, FELINE approaches achieved best results, but we need to highlight some issues:

- The INTERVAL approach does not scale well, failing for very large graphs. The problem remains in the memory requirements for building the index.

- FERRARI and TF-Label outperform GRAIL in query and construction times, but TF-Label generated the smallest index. However, with the best of our efforts, we were unable to identify the reasons that made this approach fail for some the synthetic large datasets.

- Every online search approach studied can prune negative queries in constant time and also applied the positive-cut strategy to prune some positive queries. Then the differences among their performances really come from the search done in those cases where the queries were not previously pruned or in case of false-positive queries. We show that Feline discards more branches in the search, avoiding that vertices that appear after the goal vertex in the topological orderings are expanded.

- The relatively simple FELINE approach for building indexes limits their size. The linear size of the indexes (with the number of vertices of the graphs) avoids the undesirable *a priori* thinking about the number of intervals that needs to be used in GRAIL, to optimize the query time and index size. Considering the INTERVAL method, this works even better, because the number of intervals is found dynamically. The number of intervals used by FERRARI's index is predefined like GRAIL, but some optimizations increase the index dynamically.

- FELINE and FELINE-B represent the best of two compromises: construction time and query time. FELINE achieved the best construction times for all datasets, and FELINE-B achieved the best query times, with a competitive index size.

5. CONCLUSION

Given a graph, we considered the task of efficiently answering whether there is a path connecting two arbitrary vertices. It is an important and challenging problem in many applications that handle very large graphs, like social networks, biological structures, dependence graphs in compilers and so on.

This paper presented Feline, a novel graph indexing method for the reachability problem. Inspired by the *Weak Dominance Drawing* technique, Feline is scalable and easy to implement. Experiments show that it outperforms the state of the art w.r.t. query and construction time, and also the size of the index it builds.

We are currently working on distributed, out-of-core and incremental versions of Feline. We believe that its index may be extended to support efficiently these versions.

6. ACKNOWLEDGMENTS

This work is partially supported by CNPq, Capes, Fapemig, and InWeb – INCT for the Web.

7. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 253–262, New York, NY, USA, 1989.

[2] K. Anyanwu and A. Sheth. p-queries: Enabling querying for semantic associations on the semantic web. In *In Proceedings of the Twelfth International World-Wide Web Conference*, pages 690–699. ACM Press, 2003.

[3] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1998.

[4] G. D. Battista, R. Tamassia, and I. G. Tollis. Area requirement and symmetry display of planar-upward drawings. Technical report, Providence, RI, USA, 1990.

[5] M. A. Bender, J. T. Fineman, and S. Gilbert. A new approach to incremental topological ordering. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'09, pages 1108–1115, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[6] D. Chakrabarti and C. Faloutsos. *Graph Mining: Laws, Tools, and Case Studies.* Synthesis Lectures on Data Mining and Knowledge Discovery. Morgan-Claypool Publishers, 2012.

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

[8] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 893–902, Washington, DC, USA, 2008. IEEE Computer Society.

[9] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In SIGMOD'13, pages 193–204, New York, NY, USA, 2013. ACM.

[10] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In EDBT'08, pages 193–204, New York, NY, USA, 2008. ACM.

[11] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, May 2003.

[12] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[13] I. B. Dhia. Access control in social networks: a reachability-based approach. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 227–232, New York, NY, USA, 2012. ACM.

[14] P. Eades, H. ElGindy, M. Houle, B. Lenhart, M. Miller, D. Rappaport, and S. Whitesides. Dominance drawings of bipartite graphs, 1994.

[15] M. Friedman. The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937.

[16] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In CIKM'05, pages 594–601, New York, NY, USA, 2005. ACM.

[17] M. Hollander and D. A. Wolfe. *Nonparametric Statistical Methods, 2nd Edition.* Wiley-Interscience, 2 edition, Jan. 1999.

[18] R. Jin, N. Ruan, S. Dey, and J. Y. Xu. Scarab: scaling reachability computation on large graphs. In SIGMOD'12, pages 169–180, New York, NY, USA, 2012. ACM.

[19] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Database Syst.*, 36(1):7:1–7:44, Mar. 2011.

[20] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In SIGMOD'09, pages 813–826, New York, NY, USA, 2009. ACM.

[21] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In SIGMOD'08, pages 595–608, New York, NY, USA, 2008. ACM.

[22] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, Nov. 1962.

[23] E. M. Kornaropoulos and I. G. Tollis. Weak dominance drawings and linear extension diameter. *CoRR*, abs/1108.1439, 2011.

[24] E. M. Kornaropoulos and I. G. Tollis. Overloaded orthogonal drawings. In *Proceedings of the 19th international conference on Graph Drawing*, pages 242–253, Berlin, Heidelberg, 2012. Springer-Verlag.

[25] R. Lempel and S. Moran. The stochastic approach for link-structure analysis (salsa) and the tkc effect. In *Proceedings of the 9th international World Wide Web conference on Computer networks*, pages 387–401, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

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

[27] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *Proceedings of the 21st International Conference on Data Engineering*, pages 360–371, Washington, DC, USA, 2005. IEEE Computer Society.

[28] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. FERRARI: Flexible and Efficient Reachability Range Assignment for Graph Indexing. In *ICDE'13: Proceedings of the 29th IEEE International Conference on Data Engineering*. IEEE, 2013.

[29] G. Siganos, S. L. Tauro, and M. Faloutsos. Jellyfish: A conceptual model for the as internet topology. *Journal of Communications and Networks*, 8(3):339–350, 2006.

[30] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[31] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In SIGMOD'07, pages 845–856, New York, NY, USA, 2007. ACM.

[32] J. van Helden, A. Naim, R. Mancuso, M. Eldridge, L. Wernisch, D. Gilbert, and S. J. Wodak. Representing and analysing molecular and cellular function using the computer. *Biol Chem*, 381(9-10):921–935, 2000.

[33] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In SIGMOD'11, pages 913–924, New York, NY, USA, 2011. ACM.

[34] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *in Proc. 22nd International Conference on Data Engineering*, page 75. IEEE Computer Society, 2006.

[35] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1):276–284, 2010.

[36] H. Yildirim, V. Chaoji, and M. J. Zaki. Dagger: A scalable index for reachability queries in large dynamic graphs. *CoRR*, abs/1301.0977, 2013.