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

Renê R. Veloso 1, Loïc Cerf 2, Wagner Meira Jr 2, Mohammed J. Zaki 3, 4

1Universidade Est. Montes Claros, Brazil 2Universidade Federal de Minas Gerais, Belo Horizonte MG, Brazil
3Qatar Computing Research Institute, Doha, Qatar 4Rensselaer 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 ∈ 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 preprocessing 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.

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 ⊆ V 2 its set of edges. A reachability query r(u, v) asks whether a vertex v ∈ V is reachable from a vertex u ∈ 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).

∀(u, v) ∈ V 2, v is reachable from u, denoted r(u, v), if and only if

\[
\begin{cases}
    u = v & \text{or} \ \exists (u, w) \in E \land r(w, v)
\end{cases}
\]

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 → V ′. Its time complexity is O(|V | + |E|). By using the acyclic graph G′, the answer to the reachability query from u ∈ V to v ∈ V (in G) is the same as the answer to the reachability query from scc(u) ∈ V ′ to scc(v) ∈ V ′.

![Figure 1: Reachability query tradeoffs](image-url)
(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 $\text{acc}(u)$ 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 $O(|V| \cdot |E|)$ time, and requires an $O(\log_2 I)$ time for each query.

GRAIL (Graph Reachability indexing via rAndomized In- terval 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 $I_u = [s_u, c_u]$ for each vertex $u$, where $s_u$ and $c_u$ denote the start and the end of the interval. The ending value $c_u$ is defined as $c_u = \text{post}(u)$, that is the post-order value of the vertex $u$, and the starting value $s_u$ is defined as $s_u = \min\{s_x | x \in \text{children}(u)\}$ if $u$ is not a leaf and $s_u = c_u$, 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 $O(d) + d$ intervals (for every vertex $v$ and $d$ intervals). Then, the time complexity for assigning the intervals is $O(\sum_d deg(v) \times d^2) = O(|E| \times d^2)$. 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 $TF$ means Topological Folding) recursively folds the graph. As a consequence, only a sub-graph of $G$ is traversed.

The resulting index can be represented graphically. The recursive application of this lemma and the transitivity of $\leq$ together prove the following theorem:

**Theorem 1.** $\forall (u, v) \in V^2, r(u, v) \Rightarrow u = v$ or $\exists w \in V \setminus \{u\} | i(u) \leq i(w) \land r(w, v)$.

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

**Theorem 1.** $\forall (u, v) \in V^2, r(u, v) \Rightarrow u = v$ or $\exists w \in V \setminus \{u\} | i(u) \leq i(w) \land r(w, v)$.

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

**Definition 3.** (Alg. definition of reachability).

$\forall (u, v) \in V^2, r(u, v) \Rightarrow u = v$ or $\exists (w, u) \in E | i(u) \leq i(v) \land r(w, v)$.

From a logical perspective, writing “$i(u) \leq i(v) \land r(w, v)$” instead of “$r(u, v)$” does not bring anything since $r(w, v)$ implies $i(w) \leq i(v)$ (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 $i(w) \leq i(v)$. As a consequence, only a sub-graph of $G$ is traversed.

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

1. Test whether $i(u) \leq i(v)$ and answer negatively if $i(u) \not\leq i(v)$ (by Theorem 1);
2. Otherwise search in $G$ a path from $u$ to $v$ that only passes by vertices whose indexes are $\leq i(v)$.

### 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 4.1 geometrically means that, for a given vertex $u$, we have only to check its upper-right quadrant, in order to
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., $i(b) \not\preceq i(d)$. 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 $S_0^1$ graph. In the related index, we have $i(u) \not\preceq i(v)$ 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 $S_0^1$, 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 $E \subseteq V^2$ |
| Result: $(X, Y)$, two vectors of size $|V|$ such that $V \subseteq V$, $i(v) = (X_v, Y_v)$ |

begin
  $X \leftarrow$ TopologicalOrdering($V, E$);
  $Y \leftarrow \emptyset$;
  heads $\leftarrow \emptyset, \ldots, \emptyset$;
  $d \leftarrow (0, \ldots, 0)$;
  forall the $(u, v) \in E$ do
    heads$_u \leftarrow$ heads$_u \cup \{v\}$;
    $d_v \leftarrow d_v + 1$;
    roots $\leftarrow \{v \in V \mid d_v = 0\}$;
  while roots $\neq \emptyset$ do
    $u \leftarrow$ arg max$_{v \in \text{heads}_u}(X_v)$;
    $Y \leftarrow$ $(Y, u)$;
    $Y \leftarrow$ $(Y, u)$;
    // Update $d$ and the roots set
    roots $\leftarrow$ roots $\setminus \{u\}$;
    forall the $v \in \text{heads}_u$ do
      $d_v \leftarrow d_v - 1$;
      if $d_v = 0$ then
        roots $\leftarrow$ roots $\cup \{v\}$;
end

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 selects 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 \( \text{roots} \). 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 \( \text{roots} \) 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 \( \text{roots} \), which is updated with the new roots \( f \) and \( g \), and now \( \text{roots} = \{ a, f, g \} \). Then, as \( g \) has the higher rank in \( X \), it is also removed from \( \text{roots} \) and inserted into the second position of \( Y \). Notice that \( g \) has no descendants, and, at that moment, \( Y = \{ b, g \} \) and \( \text{roots} = \{ a, f \} \). The vertex \( f \) is the next chosen and \( Y = \{ b, g, f \} \). In Figure 3, the vertices \( b, g, f \) 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 \( i(u) \neq i(v) \) 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 \( i(u) \leq i(v) \), 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 \( i(u) < i(v) \) then
5. for all \( w \in \text{heads}_u \) do
6. if Reachable(w,v) then
7. return true;
8. return false;

Let \( \tau : V \to \{1, 2, \ldots, n\} \) 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.

#### 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

---

\(^1\)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.
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 ∈ G, either u and v belong to the same strongly-connected component or the weak dominance relation does not hold (i.e., i(u) ̸< i(v)). However, when i(u) ≤ 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 happen. 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_u ⊆ I_v, 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 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] ⊆ [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 = 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 ≠ v, then l_u < l_v. Algorithm 3 applies this filter (line 6).

Figure 9 helps to understand the applicability of level fil-


\[ r(u, v) = \begin{cases} 
1 & \text{if } (u, v) \in E \\
0 & \text{otherwise} 
\end{cases} \]

\[ \text{Cluster-coeff} = \frac{2|E|}{V(V-1)} \]

\[ \text{Eff-diameter} = \text{Effective Diameter} \]

\[ \text{ECC} = \text{Effective Eccentricity} \]

\[ \text{roots} = \text{Number of Root Vertices} \]

\[ \text{leafs} = \text{Number of Leaf Vertices} \]

\[ \text{Table 1: Datasets} \]

<table>
<thead>
<tr>
<th>Graph</th>
<th>vertices</th>
<th>edges</th>
<th>Cluster-coeff</th>
<th>Eff-diameter</th>
<th>roots</th>
<th>leafs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arxiv</td>
<td>6000</td>
<td>66707</td>
<td>0.35</td>
<td>5.48</td>
<td>961</td>
<td>624</td>
</tr>
<tr>
<td>Yago</td>
<td>6642</td>
<td>42392</td>
<td>0.24</td>
<td>6.57</td>
<td>5176</td>
<td>264</td>
</tr>
<tr>
<td>Go</td>
<td>6793</td>
<td>13361</td>
<td>0.07</td>
<td>10.92</td>
<td>64</td>
<td>3687</td>
</tr>
<tr>
<td>Pubmed</td>
<td>9500</td>
<td>40028</td>
<td>0.19</td>
<td>6.83</td>
<td>2069</td>
<td>4402</td>
</tr>
<tr>
<td>citeseer</td>
<td>10720</td>
<td>44258</td>
<td>0.28</td>
<td>8.36</td>
<td>4572</td>
<td>1368</td>
</tr>
<tr>
<td>Uniprot22m</td>
<td>159544</td>
<td>159242</td>
<td>0.09</td>
<td>10.53</td>
<td>35425</td>
<td>168552</td>
</tr>
<tr>
<td>citeseer</td>
<td>640400</td>
<td>5801266</td>
<td>0.06</td>
<td>4.8</td>
<td>597149</td>
<td>574719</td>
</tr>
<tr>
<td>Go-uniprot</td>
<td>6967964</td>
<td>3470233</td>
<td>0.00</td>
<td>4.41</td>
<td>6946721</td>
<td>4</td>
</tr>
<tr>
<td>Uniprot100m</td>
<td>15917259</td>
<td>16087293</td>
<td>0.00</td>
<td>14499959</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Uniprot150m</td>
<td>26536290</td>
<td>26339555</td>
<td>0.00</td>
<td>21650005</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{Table 2: Synthetic Dataset} \]

<table>
<thead>
<tr>
<th>Graph</th>
<th>vertices</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>50M</td>
<td>50M</td>
<td>50M</td>
</tr>
<tr>
<td>60M</td>
<td>60M</td>
<td>60M</td>
</tr>
<tr>
<td>70M</td>
<td>70M</td>
<td>70M</td>
</tr>
<tr>
<td>80M</td>
<td>80M</td>
<td>80M</td>
</tr>
<tr>
<td>90M</td>
<td>90M</td>
<td>90M</td>
</tr>
<tr>
<td>100M</td>
<td>100M</td>
<td>100M</td>
</tr>
<tr>
<td>200M</td>
<td>200M</td>
<td>200M</td>
</tr>
<tr>
<td>500M-1</td>
<td>500M</td>
<td>500M</td>
</tr>
<tr>
<td>100M-5</td>
<td>100M</td>
<td>100M</td>
</tr>
<tr>
<td>100M-10</td>
<td>100M</td>
<td>100M</td>
</tr>
</tbody>
</table>

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: Arxiv7, Citeseer8, Go4, Pubmed3 and Yago9. We also used 6 large real-world DAGs (sparse and dense): Cit-Patents7, Citeseerx8, Uniprot22m, Go-uniprot, Uniprot100m and Uniprot150m9. Table 1 shows some characteristics of the graphs, such as the number of vertices and edges, clustering coefficient, effective diameter (instead of diameter) available from [35] (the authors used the SNAP software\textsuperscript{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].

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 90G ram. However, all implementations are single threaded.

4.2.1 Queries

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

![CD diagram](image)

**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.

![Query times](image)

**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 \( G' = (V, E') \), where \( E' \) 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 leaves of the DAG. Figure 12 shows plottings from four DAGs indexes: Arxiv, 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 \( G' \), 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 query times for real graphs. Average times (in milliseconds).

<table>
<thead>
<tr>
<th>Graph</th>
<th>GRAIL</th>
<th>INTERVAL</th>
<th>FERRARI</th>
<th>TF-Label</th>
<th>FELINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>13.607</td>
<td>31.409</td>
<td>34.591</td>
<td>7121.763</td>
<td>48.136</td>
</tr>
<tr>
<td>Pubmed</td>
<td>117.323</td>
<td>166.103</td>
<td>213.040</td>
<td>22575.605</td>
<td>17.403</td>
</tr>
</tbody>
</table>

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

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

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

<table>
<thead>
<tr>
<th>Graph</th>
<th>FELINE</th>
<th>FELINE-I</th>
<th>FELINE-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arxiv</td>
<td>5.533</td>
<td>5.467</td>
<td>10.760</td>
</tr>
<tr>
<td>Yago</td>
<td>4.206</td>
<td>3.776</td>
<td>8.090</td>
</tr>
<tr>
<td>Pubmed</td>
<td>5.928</td>
<td>5.465</td>
<td>10.700</td>
</tr>
<tr>
<td>Citeseer</td>
<td>7.356</td>
<td>6.821</td>
<td>14.102</td>
</tr>
<tr>
<td>Go</td>
<td>3.333</td>
<td>3.737</td>
<td>8.090</td>
</tr>
</tbody>
</table>

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 15: Index sizes for real graphs.

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

<table>
<thead>
<tr>
<th>Graph</th>
<th>FELINE-SCAR</th>
<th>GRAIL-SCAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arxiv</td>
<td>203.761</td>
<td>319.452</td>
</tr>
<tr>
<td>Yago</td>
<td>8.691</td>
<td>20.760</td>
</tr>
<tr>
<td>Polledu</td>
<td>8.703</td>
<td>17.932</td>
</tr>
<tr>
<td>citeseer</td>
<td>10.639</td>
<td>19.065</td>
</tr>
<tr>
<td>uniprot22m</td>
<td>23.038</td>
<td>63.202</td>
</tr>
<tr>
<td>Cit-patents</td>
<td>30.749</td>
<td>85.538</td>
</tr>
<tr>
<td>citeseer</td>
<td>35.248</td>
<td>74.649</td>
</tr>
<tr>
<td>Go-uniprot</td>
<td>28.025</td>
<td>91.111</td>
</tr>
<tr>
<td>uniprot150m</td>
<td>35.070</td>
<td>91.111</td>
</tr>
</tbody>
</table>

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

VAL 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 of 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


S. J. van Schauik 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.

