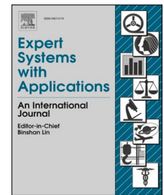
Expert Systems With Applications 181 (2021) 114962

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa



# Extended high dimensional indexing approach for reachability queries on very large graphs

Rodrigo Ferreira da Silva [a], Sebastián Urrutia [a,b,*], Lars Magnus Hvattum [b]

[a] Department of Computer Science, Federal University of Minas Gerais, Av. Antônio Carlos, 6627, Pampulha, Belo Horizonte, Minas Gerais, Brazil
[b] Department of Logistics, Molde University College, Britvegen 2, 6410 Molde, Norway

ARTICLE INFO

*Keywords:*
Directed acyclic graphs
Topological sorts
Reachability queries
Graph indexing

ABSTRACT

Given a directed acyclic graph $G = (V, A)$ and two vertices $u, v \in V$, the reachability problem is to answer if there is a path from $u$ to $v$ in the graph. In the context of very large graphs, with millions of vertices and a series of queries to be answered, it is not practical to search the graph for each query. On the other hand, the storage of the full transitive closure of the graph is also impractical due to its $O(|V|^2)$ size. Scalable approaches aim to create indices used to prune the search during its execution. Negative indices may be able to determine (in constant time) that a query has a negative answer while positive indices may determine (again in constant time) that a query has a positive answer. In this paper we propose a novel scalable approach called LYNX that uses a large number of topological sorts of $G$ as a negative cut index without degrading the query time. A similar strategy is applied regarding a positive cut index. In addition, LYNX proposes a user-defined index size that enables the user to control the ratio between negative and positive cuts depending on the expected query pattern. We show by computational experiments that LYNX consistently outperforms the state-of-the-art approach in terms of query-time using the same index-size for graphs with high reachability ratio. In intelligent computer systems that rely on frequent tests of connectivity in graphs, LYNX can reduce the time delay experience by end users through a reduced query time. This comes at the expense of an increased setup time whenever the underlying graph is updated.

## 1. Introduction

Reachability is a fundamental problem on graphs that consists of determining whether there exists a directed path from a vertex $u \in V$ to another vertex $v \in V$ in a given directed graph $G = (V, A)$. In this work we deal with this problem restricted to directed acyclic graphs (DAGs) which can be obtained from general directed graphs in linear time by collapsing the *strongly connected components* of the original directed graph by using e.g., Tarjan's classic algorithm (Tarjan, 1972).

A query can be answered in constant time if the transitive closure of the DAG has previously been computed and stored. As an alternative to the storage of a quadratic size index, a linear-time depth-first search (DFS) can be triggered for each query to be answered.

For highly queried graphs with millions of vertices and edges, triggering a graph search for each query is prohibitive. Moreover, computing the transitive closure is also prohibitive in terms of execution time and storage space. Efforts have been made to reduce the time and storage space to compute the transitive closure of the graph with the use of labeling and compression procedures (e.g. He, Wang, Yang, & Yu, 2005; Jin, Xiang, Ruan, & Fuhry, 2009; Chen, 2009; Jin, Ruan, Xiang, & Wang, 2011; Cheng, Huang, Wu, & Fu, 2013; Jin & Wang, 2013; Zhou et al., 2017; Zhou et al., 2018).

The context of very large graphs is becoming increasingly common with the emergence of massive data sets from applications on social and biological networks, software analysis and traffic routing, among others. In particular, the World-Wide Web can be seen as a huge directed graph with billions of vertices and edges (Kleinberg, Kumar, Raghavan, Rajagopalan, & Tomkins, 1999). Expert systems that rely on determining connectivity in graphs therefore benefit immensely by speeding up how fast such queries can be answered. Applications of this can be found in the area of the semantic web, or any other area where XLM graphs can be built to answer structural queries (Wang, He, Yang, Yu, & Yu, 2006), as well as for applications in biology, for example with respect to protein–protein interaction, metabolic pathways, and gene regulatory

* Corresponding author at: Department of Logistics, Molde University College, Britvegen 2, 6410 Molde, Norway.
*E-mail addresses:* rfsilva@dcc.ufmg.br (R.F. da Silva), surrutia@dcc.ufmg.br (S. Urrutia), lars.m.hvattum@himolde.no (L.M. Hvattum).

https://doi.org/10.1016/j.eswa.2021.114962
Received 8 October 2019; Received in revised form 8 March 2021; Accepted 24 March 2021
Available online 24 April 2021
0957-4174/© 2021 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

networks (Yildirim, Chaoji, & Zaki, 2012).

Recent scalable approaches (Yildirim et al., 2012; Anand, Seufert, Bedathur, & Weikum, 2013; Veloso, Cerf, & Zaki, 2014; Li, Hua, & Zhou, 2017; Wei, Yu, Lu, & Jin, 2018; Su, Zhu, Wei, & Yu, 2017) aim to index the graph consuming acceptable preprocessing time and storage space to reduce the time spent during the graph search in each query processing. These are known as refined online search methods (Jin, Ruan, Dey, & Xu, 2012) and Label + G methods (Wei et al., 2018).

Indices induce two distinct types of cuts. A negative cut index induces a set of negative cuts, i.e. a subset of all ordered pairs of vertices $(u, v) \in V \times V$ for which $v$ is not reachable from $u$. A negative cut allows to prune the search tree as soon as an intermediate vertex known for not reaching $v$ is explored. On the other hand, a positive cut index induces a set of positive cuts, i.e. a subset of all ordered pairs of vertices $(u, v) \in V \times V$ for which $u$ reaches $v$ (a subset of the transitive closure of $G$). A positive cut allows to halt the search and answer the query positively. In this setting, both negative and positive cuts are exact.

In this work we propose a novel reachability index approach called LYNX (extended high dimensional reachabiLitY iNdeX). Like FELINE (Veloso et al., 2014), LYNX uses a dimensional approach that represents its negative cut index with topological sorts of the graph. We introduce improvements over FELINE's indexing algorithm to overcome some limitations that are shown later in this article. Additionally, LYNX introduces a user-defined number of topological sorts on the negative cut index without degrading the query performance. In HD-GDD (Li et al., 2017) the authors also propose the use of many (more than two) topological sorts. However, unlike the case of LYNX, the use of a large set of topological sorts may degrade the query time. A similar strategy is applied to the positive cut index, extending the approach proposed by Yildirim et al. (2012).

Moreover, LYNX lets the user set the amount of memory available to store the indices. As more memory becomes available, the query performance increases at the cost of an increase in preprocessing time and space.

The original contributions of this work can be summarized as follows:

- Opportunities for improvement over FELINE (Veloso et al., 2014), HD-GDD (Li et al., 2017), IP (Wei et al., 2018) and BFL (Su et al., 2017) are identified in Section 4.
- In Section 5, we propose a strategy to improve FELINE's negative cut index generating diverse topological sorts resulting in fewer false positives.
- Also in Section 5, as the primary contribution of this work, we introduce a novel method called LYNX that allows a larger index size, storing a larger set of topological sorts and sets of intervals, but using a bounded number for each query without degrading query time.
- Computational results are presented in Section 6 for benchmark instances proposed by Yildirim et al. (2012) and synthetic instances with higher average vertex degree.

## 2. Preliminaries

Given an undirected graph $G = (V, E)$, where $V$ is the set of vertices and $E$ the set of edges, the *distance* between two vertices is the length of the minimum path connecting them. The *eccentricity* of a given vertex is the maximum distance to any vertex of the graph. The maximum eccentricity represents the *diameter* of the graph. Vertices with eccentricity equal to the diameter are on the *periphery*, while vertices with minimum eccentricity (*radius*) are in the *center of the graph*.

A simple directed acyclic graph (DAG) is represented as $G = (V, A)$ where $V$ is the set of vertices and $A$ the set of arcs between them. The set of *successors* of a vertex $u$ is represented by $Suc(u) = \{v \in V | (u, v) \in A\}$ and the set of *predecessors* is represented by $Pre(u) = \{v \in V | (v, u) \in A\}$. Then, we can define the *indegree* and *outdegree* of a given vertex $u$ respectively denoted by $d_{in}(u) = |Pre(u)|$ and $d_{out}(u) = |Suc(u)|$. A vertex $u$ with no outgoing arcs, i.e. $d_{out} = 0$, is called a *sink*, and a vertex $u$ with no incoming arc, i.e. $d_{in} = 0$, is called a *source*.

A *topological sort* $t$ of $G$ is permutation of $V$ such that $\forall (u, v) \in A$ it holds that $t(u) < t(v)$, where $t(u)$ represents the position of $u$ in $t$. That is, $t$ is a bijective function that maps each vertex of $G$ to a number in $\{1, \ldots, |V|\}$ with the additional property that $t(u)$ is smaller than $t(v)$ for every edge $(u, v) \in A$. A *reachability query* from $u$ to $v$ in $G$ is denoted by $u \overset{?}{\rightsquigarrow} v$. If $v$ is reachable from $u$ by transversing the arcs of the graph, then the query is answered positively, represented by $u \rightsquigarrow v$. Otherwise, it is represented by $u \neg\rightsquigarrow v$.

We can use this notation to represent the set of all vertices that can reach $u$ as $In(u) = \{v \in V | v \rightsquigarrow u\}$ and the set of all vertices reached by $u$ as $Out(u) = \{v \in V | u \rightsquigarrow v\}$.

Negative and positive cut indices can be used to prune a search during a reachability query in the graph. A *negative cut index* induces a subset of all non-reachable pairs of vertices in $G$, denoted by $I_N \subseteq \{(u, v) \in V \times V | u \neg\rightsquigarrow v\}$. A *positive cut index* induces a subset of all reachable pairs in the graph, denoted by $I_P \subseteq \{(u, v) \in V \times V | u \rightsquigarrow v\} = tr(G)$ where $tr$ represents the *transitive closure* of $G$.

## 3. Related work

Proposed in 2010, GRAIL (Yildirim et al., 2012) is considered the first approach for reachability queries that scales to very large real-world graphs. In its preprocessing stage, GRAIL builds three indices: a negative-cut index, a positive-cut index and a topological-level filter as an additional negative cut index. The first negative cut index consists of an interval for each vertex of the graph. The interval set $L$ is built in linear time by a randomized *min-post* labeling algorithm. For each pair of vertices $u, v \in V$ if $L_v \neg\subseteq L_u$, then $u \neg\rightsquigarrow v$. However, if $L_v \subseteq L_u$, then either $u \rightsquigarrow v$ or $(u, v)$ is considered to be a false positive pair of the index.

Similarly, the positive cut index is an assignment of an interval to each vertex of the graph constructed by a *min-post* labeling algorithm that produces an interval set representing a sub-tree (set of paths) of the DAG, such that if $L_v \subseteq L_u$, then $u \rightsquigarrow v$.

In the topological level filter, vertices are divided into levels such that all sinks (vertices with no outgoing arcs) are placed in the first level and the other vertices are placed one level above the level of the highest of its successors. Then, the topological level filter can be applied as a negative cut index during the search, since if the level of $u$ is smaller than or equal to the level of $v$ then $u \neg\rightsquigarrow v$.

FERRARI (Anand et al., 2013) introduces some flexibility over GRAIL, that gives more control to the user to set a bound to the size of its positive cut index, which has a direct effect on the query processing time. In addition, it uses a seed based pruning approach, that computes the reachability of vertices with high outdegree to every other vertex of the graph. For a query $u \overset{?}{\rightsquigarrow} v$, if a seed reached by $u$ reaches $v$, then $u \rightsquigarrow v$. However, if a seed reaches $u$ and does not reach $v$, then $u \neg\rightsquigarrow v$.

As an alternative to GRAIL and FERRARI, FELINE (Veloso et al., 2014) proposes an index representation inspired by *Dominance Drawing*. In this approach, the negative cut index is represented by two topological sorts $t_X$ and $t_Y$. The first topological sort, is generated by using a DFS algorithm, in which each vertex is added to the end of $t_X$ as all its successors are explored. The topological sort $t_Y$ is generated using the *Maximum-Rank* heuristic, proposed by Kornaropoulos (2012), as shown

in Algorithm 1.

**Input:** $G$, $t_X$

**Output:** $t_Y$

1 Initialize $S_G$ ;
2 $t_Y \leftarrow Nil$ ;
3 **for** $i \leftarrow 1$ *to* $|V|$ **do**
4 $\quad u \leftarrow maxRank_{t_X}(S_G)$ ;
5 $\quad t_Y[i] = u$ ;
6 $\quad G \leftarrow G - u$ ;
7 $\quad$ Update $S_G$ ;
8 **end**

The algorithm starts by initializing $S_G$ as the set of sources. In each iteration, the vertex of $S_G$ with the highest position on $t_X$ (*maxRank* function) is chosen to be added on $t_Y$. That vertex is then removed from $G$ in line 6. Consequently, $S_G$ is updated by removing $u$ and adding new sources introduced in $G$, if any, in line 7. The algorithm finishes when $t_Y$ contains all vertices of $G$.

During the processing of a query $u \overset{?}{\rightsquigarrow} v$, on an intermediary vertex $w$ such that $u \rightsquigarrow w$, if $t_X(v) < t_X(w) \vee t_Y(v) < t_Y(w)$, then $w \neg\rightsquigarrow v$ and the search can be pruned on $w$. Therefore, this negative cut index is able to cut each ordered pair in $I_N = \{(u, v) \in V \times V | t_X(v) < t_X(u) \vee t_Y(v) < t_Y(u)\}$.

Maximizing the size of $I_N$ is equivalent to maximizing the set of inversions between $t_X$ and $t_Y$, that can be written as $\bar{I}(t_X, t_Y) = \{(u, v) \big| t_X(u) < t_X(v) \wedge t_Y(v) < t_Y(u)\}$. Alternatively, it is also equivalent to minimizing the size of the set of intersections $I(t_X, t_Y) = \{(u, v) | t_X(u) < t_X(v) \wedge t_Y(u) < t_Y(v)\}$, where each pair $(u, v)$ are in the same relative order in both $t_X$ and $t_Y$.

HD-GDD (Li et al., 2017) is an extension of FELINE that works with three or more topological sorts on its negative cut index. With more topological sorts, it is possible to introduce more inversions between pairs of vertices, which reduces the number of vertices explored during the search. However, each topological sort on the index may need to be checked for each vertex explored. Then, there is a trade-off between number of vertices explored and evaluation time spent on each vertex.

Wei et al. (2018) proposed a random labeling approach called IP, based on independent permutations (Broder, 1997). The IP algorithm uses the fact that if $u \rightsquigarrow v$, then $In(u) \subseteq In(v)$ and $Out(v) \subseteq Out(u)$. Therefore, if $In(u) \neg\subseteq In(v)$ or $Out(v) \neg\subseteq Out(u)$, then $u \neg\rightsquigarrow v$.

The algorithm generates random permutations of the vertices by using an unbiased algorithm. For each random permutation $\pi$, the algorithm generates two sets $\mathscr{L}_{in}(u)$ and $\mathscr{L}_{out}(u)$. The set $\mathscr{L}_{in}(u)$ represents the set of top-$k$ smallest numbers $\pi(v)$ such that $v \in In(u)$. Similarly, $\mathscr{L}_{out}(u)$ is the set of top-$k$ smallest numbers $\pi(v)$ such that $v \in Out(u)$. The index is generated for all vertices $u \in V$ in time $O(k(|V| + |A|))$ with total index size at most $2kn$. A DFS search from an intermediate vertex $u$ to $v$ can be pruned negatively if $min(\mathscr{L}_{in}(v)) > min(\mathscr{L}_{in}(u)) \vee min(\mathscr{L}_{out}(v)) > min(\mathscr{L}_{out}(u))$.

IP also considers two additional labels: two types of topological level filter based on the one proposed in GRAIL (Yildirim et al., 2012), one starting from sinks and another starting from the sources of the graph; and a huge vertex label index. This last label index is generated by computing the transitive closure of the vertices with high degree in the graph and used to prune the DFS search whenever a vertex with high outdegree is reached.

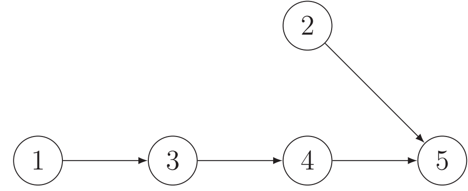



**Fig. 1.** *Maximum-Rank* pathological case.

Bloom filter Labeling (BFL) (Su et al., 2017) proposes another labeling approach that uses randomness to create its index. Similarly to IP, BFL explores the fact that if $In(u) \neg\subseteq In(v)$ or $Out(v) \neg\subseteq Out(u)$, then $u \neg\rightsquigarrow v$. Briefly, each vertex is mapped to a number in the subset $\{1, 2, \ldots, s\}$ by a hash function $g(\cdot)$. Each vertex $u \in V$ is associated with a subset of $\{1, 2, \ldots, s\}$ such that $\mathscr{L}_{out}(u) = \bigcup_{(u,w) \in A} \mathscr{L}_{out}(w)$. If $\mathscr{L}_{out}(v) \neg\subseteq \mathscr{L}_{out}(u)$, then $u \neg\rightsquigarrow v$ and the DFS search can be pruned at this point. Otherwise, either $u \rightsquigarrow v$ or the pair is a false positive.

BFL uses a bit vector to represent a subset of $\{1, 2, \ldots, s\}$ for each vertex. Then, the unions to compute each subset $\mathscr{L}_{out}$ of the index can be done efficiently by performing a bitwise-or on two bit vectors. This same strategy is applied during the query search to check if $\mathscr{L}_{out}(v) \neg\subseteq \mathscr{L}_{out}(u)$ efficiently and prune the DFS search.

## 4. Opportunities for improvement

In this section we identify some opportunities for improvement over existing literature on reachability approaches for very large graphs.

### 4.1. FELINE

Fig. 1 shows a pathological case for the *Maximum-Rank* heuristic used by FELINE. Let $G = (V, A)$ be the DAG presented in Fig. 1 and $t_X = (1, 2, 3, 4, 5)$ a topological sort of $G$. The result obtained by *Maximum-Rank* given $G$ and $t_X$ as input is $t_Y = (2,1,3,4,5)$. Note that $\bar{I}(t_X, t_Y) = \{(1, 2)\}$, meaning that only one inversion between $t_X$ and $t_Y$ was obtained. A better choice would be $t'_Y = (1, 3, 4, 2, 5)$ giving the inversion of pairs (2, 3) and (2, 4).

This simple example can be extended to show that the ratio between the number of inversions obtained by the heuristic and the maximum number of inversions that may be obtained can be arbitrarily large, by introducing new vertices extending the path from 1 to 5 in $G$, and modifying $t_X$ accordingly.

The *Maximum-Rank* heuristic used by FELINE attempts to solve the Weak Dominance Drawing (WDD) (Kornaropoulos & Tollis, 2012) problem. The optimization version of WDD consists of finding two topological sorts, $t_X$ and $t_Y$, of a given DAG $G = (V, A)$, minimizing the intersection $I(t_X, t_Y)$ between them. Minimizing the size of the intersection set is the same as maximizing the number of pair inversions $\bar{I}(t_X, t_Y)$.

However, since FELINE fixes the topological sort $t_X$ at the start of the index generation process, the quality of the solution it can find is bounded by the number of inversions between $t_X$ and another topological sort of the graph. This number may be much smaller than the maximum number of inversions between two arbitrary topological sorts of $G$. In this way, considering $t_X$ as a parameter of the problem instead of a variable, the solution search space is dramatically reduced and in consequence the quality of the best solution to be found may be reduced.

This restricted version of WDD, in which one topological sort is fixed as an input, is called One-Sided Weak Dominance Drawing (OSWDD) (de Silva, Urrutia, & dos Santos, 2019) and was proven to be NP-hard.

### 4.2. HD-GDD

HD-GDD (Li et al., 2017) extends the dimensional approach of FELINE to use 3 or more topological sorts on its index. The first topological sort is generated with a DFS algorithm, as it is done in FELINE. Then, the algorithm computes the higher dimensions using a *Maximum-Rank*-like heuristic, however changing the function *maxRank* to iteratively select the sources using heuristic conditions that consider, in this order: the vertex with maximum coordinate in previous topological sorts; the vertex with maximum total sum of the positions on previous topological sorts; the deviation in the topological sorts positions; and, the minimum position in previous topological sorts, when the previous conditions are all tied.

Each new topological sort introduced into the index has to be examined for every vertex explored during the DFS search. Then, if a topological sort introduces just a few inversions in comparison with the previous ones, the time needed to check each vertex during the search is increased without pruning the search on a significant number of vertices, which may increase the query time. Consequently, HD-GDD approach works well only using up to 5 topological sorts.

### 4.3. IP and BFL

Even though IP and BFL have significantly outperformed the previous approaches reducing the query and indexing time, considerable improvements were made only in the negative cut index. The huge vertex label of IP can be seen as a modest positive cut index since the number of selected vertices is relatively small in comparison with the graph size (only vertices with outdegree $>100$). This implies that its performance can be degraded for positive queries.

BFL is considered the state-of-the-art approach for reachability queries on very large DAGs. It uses a simple positive cut index similar to GRAIL, however considering only one graph *min-post* transversal labeling. Since 2010, little effort has been made to build a better index suitable for positive queries, even though they may represent a significant part of queries in realistic scenarios.

## 5. LYNX

In this section, we propose a novel approach for the reachability problem on very large graphs called LYNX. The negative cut index is based on FELINE, however we introduce a new simple but effective method to improve the negative cut index generation, called jumps, that works together with FELINE's *Maximum-Rank* heuristic. Additionally, we propose another simple but powerful improvement in the index generation and query answering procedure. As in HD-GDD, we use a larger set of topological sorts but, opposite to HD-GDD, our approach does not degrade the performance of each individual query since only two topological sort positions are compared in each query as in FELINE.

The same strategy used in the negative cut index is also used on the positive cut index to allow a larger family of graph interval label sets without degrading query performance. With this, we significantly improve on GRAIL's original strategy that is widely used in almost all latter approaches for the problem (e.g. Anand et al., 2013; Veloso et al., 2014; Li et al., 2017; Su et al., 2017).

Together with the strategies briefly described above, LYNX proposes a flexible approach in which the user can define a memory budget for the whole index and also set the portion of that budget that it is allocated to the negative and positive indices. Then, the index can be boosted using user knowledge about the dataset or query patterns.

### 5.1. Graph of topological sorts

We define the graph of topological sorts of a DAG as $G^{TS} = (V^{TS}, E^{TS})$. The vertex set $V^{TS}$ represents the set of all topological sorts of $G$, such that $V^{TS} = \{\text{permutation } p \text{ of } V | (u,v) \in A \Rightarrow p(u) < p(v)\}$. The edge set is defined as $E^{TS} = \{(s,t) \in V^{TS} \times V^{TS} | |\bar{I}(s,t)| = 1\}$. Then, there exists an edge between every pair $(s,t)$ of topological sorts of $G$ for which the number of inversions between them is exactly one. This means that two adjacent topological sorts differ only in the position of two consecutive vertices that have been swapped. This class of graphs is extensively studied in Brightwell and Massow (2013).

The eccentricity of a vertex $p \in V^{TS}$ can be seen as the minimum number of consecutive swap moves necessary to change $p$ into any other topological sort of $G$. The periphery of $G^{TS}$ is the set of those topological sorts having maximum eccentricity, i.e. those which needs the largest number of swap moves to reach the farthest topological sort. This number of swap moves equals the diameter of $G^{TS}$.

From the point of view of the graph of topological sorts, *Weak Dominance Drawing* (WDD) corresponds to finding a diametral pair $(p_1, p_2)$ on $G^{TS}$. This problem was proven to be NP-hard in Kornaropoulos and Tollis (2011).

Corneil, Dragan, and Köhler (2002) show an approximate algorithm to compute the diameter of graphs with no induced cycles of size larger than $k$. The proposed algorithm performs two consecutive BFS procedures: the first one starting from an arbitrary vertex and storing the last vertex visited $u$; the second one starts from $u$. Let $v$ be the last vertex visited in the second BFS, then the returned pair is $(u,v)$. The authors showed that the distance between $u$ and $v$ is at least $diam(G) - \lfloor k/2 \rfloor. -2$.

Therefore, the diameter of $G^{TS}$ can be approximated by $diam(G^{TS}) - 5$ since, as shown by Massow (2009), the cycle space of $G^{TS}$ is generated by only 4-cycles and 6-cycles. However, executing BFS on $G^{TS}$ is not practical since the number of vertices of that graph, i.e. the number of topological sorts of $G$, can be exponentially large.

Executing BFS on $G^{TS}$ to obtain one vertex with maximum distance from the start vertex is equivalent to solve an instance of the One-Sided Weak Dominance Drawing Problem (OSWDD) (de Silva et al., 2019), an NP-hard problem.

### 5.2. Jumps strategy

As shown in Section 4.1, the *Maximum-Rank* heuristic does not attempt to solve WDD. Since one of its topological sorts is fixed, the problem tackled by *Maximum-Rank* can be seen as finding a vertex furthest away from a given vertex in $G^{TS}$ which corresponds to solve OSWDD. Independently of the quality of the heuristic, the quality of its solution (the number of inversions between $t_X$ and $t_Y$) is bounded by the eccentricity of the vertex $t_X$ in $G^{TS}$.

Therefore, we propose a jumps heuristic based on the idea of the algorithm proposed by Corneil et al. (2002) that uses two BFS to obtain an approximation to diametral pairs on $G^{TS}$ as shown in Section 5.1. However, since $G^{TS}$ can be exponentially large, we use the *Maximum-Rank* heuristic on $G$ as an alternative to BFS on $G^{TS}$. This is a greedy heuristic to find a furthest vertex on $G^{TS}$ from another given vertex.

We start with the jumps strategy generating a random topological sort $t_X$. For this, we use an algorithm similar to the *Maximum-Rank* heuristic, replacing *maxRank* function with a random vertex selection from $S_G$ (see Algorithm 1). Note that the topological sort $t_X$ represents a vertex of $G^{TS}$.

After this, we use the *Maximum-Rank* heuristic to generate $t_Y$. At this point we have two topological sorts, $t_X$ and $t_Y$, the first one created without any optimization criterion (random) and the second one generated attempting to maximize its distance from the first one.

At this point, the *Maximum-Rank* heuristic is executed again with $t_Y$ as input and $t'_X$ as outcome. The distance between $t_X$ and $t_Y$ is probably smaller than the distance between $t_Y$ and $t'_X$. Indeed, if we replace the heuristic by a BFS algorithm on $G^{TS}$ we would be certain that the distance between $t_Y$ and $t'_X$ would be at least as large as the distance between $t_X$ and $t_Y$ since the maximum distance between $t_Y$ and another

vertex of $G^{TS}$, which BFS computes, is bounded from below by the distance between $t_Y$ and $t_X$.

It is important to note that $t'_X$ is always a local optimum within the swap neighborhood. In each iteration, the *Maximum-Rank* heuristic keeps track of all sources of the DAG and a source with maximum ranking on $t_Y$ is removed from the graph. Then, if $u$ is next to $v$ in $t'_X$ such that $t'_X = (\ldots,u,v,\ldots)$, then $u \rightsquigarrow v$ or they are reversed on $t_Y$.

### 5.3. LYNX Negative cut index

The negative cut index of LYNX is generated using the jumps strategy. Both $t'_X$ and $t_Y$ produced by an execution of the proposed heuristic are added to the index. The first pair $t'_X$ and $t_Y$ of topological sorts is generated by creating $t_X$ with a DFS-like algorithm as starting point, and next ones with $t_X$ as a random topological sort as described in Section 5.2.

The reachability query proposed by HD-GDD checks each topological sort in the index for every vertex visited during the DFS execution. LYNX uses a different strategy. For each vertex $u$, LYNX saves the topological sort where $u$ has the highest position, defined as $t_{H(u)}$, and the topological sort where the vertex has the lowest position among all, defined as $t_{F(u)}$. This can be done in $O(|V|N)$ where $N$ is the number of topological sorts generated.

With this procedure, we can efficiently prune the search using the negative cut index by checking only two topological sorts for each vertex explored. For a query $u \overset{?}{\rightsquigarrow} v$, if $t_{H(u)}(u) > t_{H(u)}(v)$ or $t_{F(v)}(u) > t_{F(v)}(v)$, the search can be pruned at this point.

Since $u$ has its highest position in $t_{H(u)}$ in comparison with all other stored topological sorts, more vertices are on the left side of $u$ in $t_{H(u)}$ and a query $u \overset{?}{\rightsquigarrow} v$ can be pruned on $u$ if $t_{H(u)}(v) < t_{H(u)}(u)$. Similarly, $t_{F(v)}$ has the highest probability to prune a search as a negative cut index for a query to $v$, since $v$ has lower position in $t_{F(v)}$.

### 5.4. LYNX positive cut index

As in GRAIL, the positive cut index of LYNX is composed by a number of assignments of intervals to each vertex. Those assignments are built by randomized DFS transversals of $G$. In each DFS transversal an interval is computed for each vertex representing a directed sub-tree of the DAG, such that if $L_v \subseteq L_u$, then $u \rightsquigarrow v$.

In GRAIL, each assignment of intervals in the index is checked for each vertex explored during a reachability query in search of a positive cut. We propose a strategy (similar to LYNX negative cut index strategy) in which the size of the interval (i.e. number of vertices inside the interval) assigned to each vertex is computed during each of the DFS transversals. Then, for each vertex $u$ we store the index of the interval assignment in which the interval of $u$ is larger. Such interval is called $p_u$.

During the DFS search for a reachability query $u \overset{?}{\rightsquigarrow} v$, for any intermediate vertex $w$ explored, if $p_w(v) \subseteq p_w(w)$ the search is halted and the query is answered positively.

### 5.5. Reachability query

Algorithm 2 depicts the DFS search function used in LYNX to answer reachability queries. The negative cut index is checked in line 4 to prune the search if the position of $u$ is higher than $v$ in $t_{H(u)}$ or $t_{F(v)}$. Otherwise, if the search is not pruned, the positive cut index is used to answer the query positively if $p_u(v) \subseteq p_u(u)$. If not, the search continues visiting $u$ neighbors.

We now show an example execution of LYNX negative cut index construction and of its usage in a reachability query for the graph in

```
1  Reachable(u, v, G)
2      if u = v then
3          return True //u ⇝ v
4      if t_H(u)(u) > t_H(u)(v) ∨ t_F(v)(u) > t_F(v)(v) then
5          return False //u ⇝̸ v
6      if p_u(v) ⊆ p_u(u) then
7          return True //u ⇝ v
8      for n ∈ Adj(u) do
9          if n has not been visited then
10             if Reachable(n, v, G) then
11                 return True
12     return False
```

Fig. 2. For this example, we define that the index to be constructed consists of four topological sorts. Then, we construct two pairs of topological sorts using the jumps strategy each of them starting from a different random permutation of the vertices of the graph. In this case: $p_1 = (4,6,5,2,1,3)$ and $p_2 = (2,4,3,1,6,5)$.

The first topological sort $t_1$ of the graph is constructed using the Maximum-Rank heuristic on the given graph with $p_1$ as parameter. $S_G = \{1,2,3\}$ which are the sources of the graph. Since 3 is the vertex most to the right in $p_1$ we add 3 to $t_1$ and remove it from the graph. After this we have $t_1 = (3)$ and $S_G = \{1,2\}$. Since 1 is more to the right than 2 in $p_1$ we continue adding 1 to $t_1$ and removing it from the graph. We repeat the above procedure until $S_G = \varnothing$. When the heuristic terminates we have $t_1 = (3,1,2,5,6,4)$.

The Maximum-Rank heuristic is applied again using $t_1$ as parameter and obtaining $t_2 = (2,1,4,3,6,5)$. Finally, following the jumps strategy, we apply the Maximum-Rank heuristic once again using $t_2$ as parameter and obtain $t_3 = (3,1,5,2,6,4)$. Then, we add $t_2$ and $t_3$ to the negative cut. Fig. 3 shows the three topological sorts constructed in the graph of topological sorts of $G$ and the jumps from $t_1$ to $t_2$ and from $t_2$ to $t_3$.

Repeating the procedure using permutation $p_2$ as parameter we obtain $t_4 = (1,3,5,2,6,4), t_5 = (2,3,6,1,4,5)$ and $t_6 = (1,3,5,2,4,6)$.

Then, the negative cut index is composed by the following topological sorts:

- $t_2 = (2,1,4,3,6,5)$,
- $t_3 = (3,1,5,2,6,4)$,
- $t_5 = (2,3,6,1,4,5)$,
- $t_6 = (1,3,5,2,4,6)$.

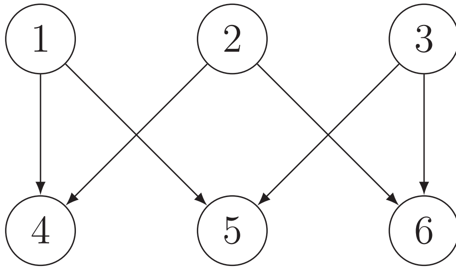



**Fig. 2.** Crown graph.

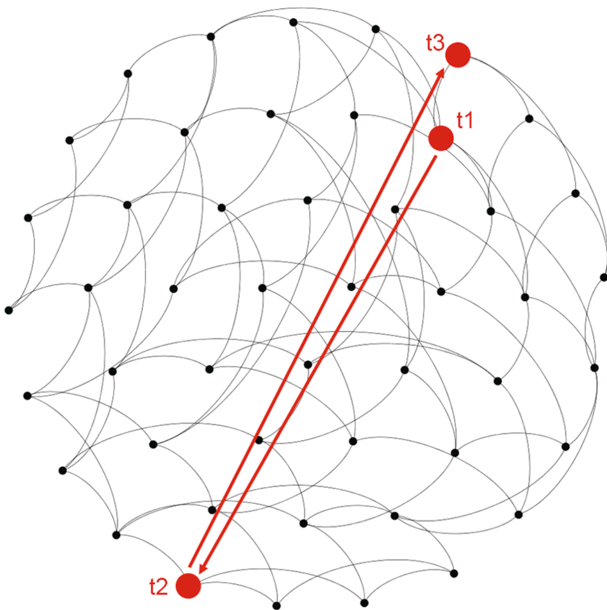



**Fig. 3.** Topological sorts $t_1, t_2$ and $t_3$ in $G^{TS}$.

For each vertex we set $tl$ as the topologial sort in which the vertex appear most to the left and $th$ as the topologial sort in which the vertex appear most to the right. In case of ties we choose the lowest indexed topological sort. The assignment is as following:

- $tl_1 = t_6$ and $th_1 = t_5$,
- $tl_2 = t_2$ and $th_2 = t_3$,
- $tl_3 = t_3$ and $th_3 = t_2$,
- $tl_4 = t_2$ and $th_4 = t_3$,
- $tl_5 = t_3$ and $th_5 = t_2$,
- $tl_6 = t_5$ and $th_6 = t_6$.

To answer query $1 \overset{?}{\rightsquigarrow} 3$ we check the topological sort in which 1 appears most to the right ($th_1 = t_5$) and the one in which 3 appears most to the left ($tl_3 = t_3$). Since $t_5(3) < t_5(1)$ we can cut the search and answer the query negatively.

### 5.6. Additional improvements

In preprocessing time, we build two Boolean vectors to represent which vertices are sources and sinks of the graph. This can be done in linear time by checking all arcs of the graph. Then, this simple index is used before examining the negative and positive cut index on reachability query. The search can be pruned whenever $u \neq v$ and $u$ is a sink or $v$ is a source of the graph.

It is important to note that this check does not add any new cut in the search. However, it improves the performance for all benchmark instances tested since sources and sinks usually represent a considerable portion of the vertices. Additionally, when this condition is verified, the search can be pruned without even the need to check indices and adjacency lists.

## 6. Experimental results

To demonstrate the efficiency of LYNX, computational experiments were conducted comparing this new approach to FELINE and BFL. BFL is the current state-of-the-art as it outperforms all the previous approaches for most benchmark instances (Yildirim et al., 2012; Anand et al., 2013; Veloso et al., 2014; Wei et al., 2018). All experiments were executed on an Intel Core i7-4790 K CPU 4.00 GHz, with 16 GB of RAM and Ubuntu 12.04. The source code was compiled with g++ with -O3 option. The original source codes of FELINE and BFL were kindly provided by the authors Veloso et al. (2014) and Su et al. (2017), respectively.

In the experiments, we focus on the time to answer queries as well as the time used for preprocessing, to make the system ready to answer queries. From the perspective of an end user of a system that relies on answering connectivity queries in graphs, the query times influences the real-time delay experienced when using the system, whereas the preprocessing time influences the setup time incurred when the database used to generate the graph is updated. These two criteria are therefore both important, but for most applications, we expect that the query time is more critical than the preprocessing time.

### 6.1. Benchmark instances

Two sets of instances were used as benchmarks, one with real-world graphs and the other with synthetic graphs. The first set, shown in Table 1, is composed of 14 real-world instances extensively used in the literature proposed by Jin et al. (2009) and Yildirim et al. (2012), from a wide spectrum of domains. The number of vertices range from 6,000 to

**Table 1**
Real-world benchmark instances.

| Instance | $\|V\|$ | $\|E\|$ | $d_{avg}$ | Sources | Sinks |
|---|---|---|---|---|---|
| arXiv | 6000 | 66,707 | 11.1 | 961 | 624 |
| go | 6793 | 13,361 | 2.0 | 64 | 3087 |
| pubmed | 9000 | 40,028 | 4.4 | 2,609 | 4702 |
| citeseer | 693,947 | 312,282 | 0.5 | 613,497 | 381,665 |
| uniprotenc_22m | 1,595,444 | 1,595,442 | 1.0 | 556,158 | 2 |
| cit-Patents | 3,774,768 | 16,518,947 | 4.4 | 515,785 | 1,685,423 |
| citeseerx | 6,540,401 | 15,011,260 | 2.3 | 567,151 | 5,740,712 |
| go_uniprot | 6,967,956 | 34,770,235 | 5.0 | 6,946,003 | 286 |
| govwild | 8,022,880 | 23,652,610 | 2.9 | 1,302,461 | 5,189,465 |
| uniprotenc_100m | 16,087,295 | 16,087,293 | 1.0 | 14,598,960 | 2 |
| yago | 16,375,503 | 25,908,132 | 1.6 | 3,003,181 | 13,372,794 |
| twitter | 18,121,168 | 18,359,487 | 1.0 | 3,138,961 | 16,383,480 |
| web-uk | 22,753,644 | 27,221,332 | 1.2 | 10,826,445 | 16,136,119 |
| uniprotenc_150m | 25,037,600 | 25,037,598 | 1.0 | 21,650,057 | 2 |

25,037,600 with the average vertex degree ranging from 0.5 to 11.

Instances `arXiv`[1], `citeseer`[2] and `citeseerx`[3] include bibliographic citations of scientific papers. Instances `go`, `go_unipprot` and `uniprotenc_150m`[4] represent knowledge databases from the Gene Ontology Project, and instances `uniprotenc_22m` and `uniprotenc_100m` are subgraphs of instance `uniprotenc_150m`. The instance `pubmed`[5] includes bibliographic citations of biomedical literature. Instance `cit-Patents`[6] comprises all US patents granted between 1975 and 1999. Finally, instances `govwild`[7], `yago`[8], `twitter` (Cha, Haddadi, Benevenuto, & Gummadi, 2010) and `web-uk`[9] are large web graph databases, ranging from social networks to knowledge databases.

We use synthetic benchmark instances to show how the performance on query time is affected as the number of arcs is increased in the graph. This set of instances consists of 20 DAGs with 10 million vertices, with the average vertex outdegree between 1 and 20. It is important to note that a synthetic instance with more arcs is not an extension of another with less arcs.

The algorithm proposed by Yildirim et al. (2012) is used to generate synthetic instances. It starts by creating a random permutation of the vertices. Then, for each arc to be added, two vertices are randomly picked and an arc is created from the leftmost to the rightmost in the generated permutation.

### 6.2. Parameters setting

In this Section we discuss about the parameters setting for BFL and LYNX for the next experiments. FELINE has no parameters to be set. Su et al. (2017) study best values for the two parameters of BFL: the size of the interval $s$ and the number of intervals $d$. They show that $s = 160$ and $d = 10$ are good values for dense graphs like `cit-Patents`.

To compare with LYNX, we started with the mentioned BFL parameters values, however they produced only modest results for BFL considering all instances tested. Experimental test showed that if we increase $s$, the overall performance of BFL is improved and the query time is reduced for instance `cit-Patents`. Note that `cit-Patents` is the same instance used as benchmark in Su et al. (2017) to improve parameters values.

Then, we use $s = 1280$ for BFL since it is the maximum value for $s$ that does not exceed the overall memory available in the test environment considering all benchmark instances. We also observed that the difference between the overall memory used by BFL and the estimated index size by the algorithm's output is much larger than what is expected for the graph in-memory representation, which suggests that this measure is not precise. For this reason, we use the overall memory usage as measure for all approaches compared. The memory limit of LYNX is set equal to the memory actually used by BFL for each instance. In this way, the comparison might be unfair for LYNX since it is limited to the amount of memory used by BFL but it is not unfair to BFL that is only constrained by the availability of physical memory in the computational environment. Also, we fix the ratio between negative and positive cut indexes to 3 for all instances tested. That is, $3/4$ of the available memory is used for the negative index and the other $1/4$ is used for the positive index.

### 6.3. Jumps strategy efficiency

In this section, we analyse the efficiency of the jumps strategy proposed for LYNX. To accomplish this, we modified LYNX to work like FELINE, with only two topological sorts on its negative cut index. With this FELINE version of LYNX as a common base, we create another version just including the jumps strategy used in LYNX.

Table 2 shows the number of negative cuts in the first level of the DFS search for 1 million of queries. Each negative cut in the beginning of the DFS avoids several vertices to be searched during the graph transversal. The results shows that the jumps strategy consistently increases the number of negative cuts for almost all graphs tested. The only exception

**Table 2**
Influence of jumps strategy on negative cuts.

| Instance | Negative cuts without jumps | Negative cuts with jumps |
|---|---|---|
| arXiv | 597,761 | 606,196 |
| go | 726,032 | 753,032 |
| pubmed | 731,600 | 768,131 |
| citeseer | 955,619 | 974,621 |
| uniprotenc_22m | 990,420 | 999,932 |
| cit-Patents | 661,565 | 686,280 |
| citeseerx | 800,270 | 812,599 |
| go_uniprot | 997,803 | 997,904 |
| govwild | 911,708 | 911,769 |
| uniprotenc_100m | 967,640 | 996,989 |
| yago | 980,250 | 980,250 |
| twitter | 853,233 | 854,114 |
| web-uk | 847,728 | 848,030 |
| uniprotenc_150m | 951,356 | 998,352 |

[1] http://arxiv.org/
[2] http://citeseer.ist.psu.edu
[3] http://citeseerx.ist.psu.edu
[4] http://www.geneontology.org/
[5] http://www.pubmedcentral.nih.gov/
[6] http://www.snap.stanford.edu/
[7] http://govwild.hpi-web.de/project/govwild-project.html
[8] http://www.mpi-inf.mpg.de/yago-naga/yago/
[9] http://law.di.unimi.it/

is the instance `yago` for which the number remained the same.

We also highlight the results for instance `uniprotenc_150m`, the largest instance tested in terms of the number of vertices, for which 95.1 % of the queries can be answered in the first level of the DFS without using jumps strategy. However, when we turned on the jumps strategy more than 99.8 % of the queries can be answered without transversing the graph.

### 6.4. Efficiency of LYNX including more topological sorts

This next experiment shows how LYNX behaves with respect to the query time, preprocessing time and index size, as we added more topological sorts in the negative cut index. In this test, we use the instance `cit-Patents`, which is one of the most difficult instances from the real-world benchmark instance set. The topological sorts are inserted in the index in pairs, and the pairs are generated by using the jumps strategy.

Table 3, shows that, preprocessing time and index size grow linearly with the number of topological sorts used in the index. On the other hand, unlike HD-GDD, LYNX can store more topological sorts in its index and each topological sort introduced improves the index reducing the query time. The query time is reduced from 17 s with two topological sorts, to 94.22 ms for 800 topological sorts. However, for 800 topological sorts, the index size grows to 12 GB in memory and the preprocessing time exceeds 18 min.

**Table 3**
Influence of the number of topological sorts on LYNX.

| Topological sorts | Index size (MB) | Preprocessing (s) | Query time (ms) |
|---|---|---|---|
| 2 | 656 | 5.1 | 17491.40 |
| 4 | 685 | 8.0 | 5070.33 |
| 6 | 713 | 10.5 | 2636.61 |
| 8 | 742 | 13.2 | 1963.98 |
| 10 | 771 | 16.5 | 1676.95 |
| 20 | 915 | 30.2 | 930.94 |
| 30 | 1059 | 45.0 | 698.12 |
| 40 | 1203 | 58.4 | 547.51 |
| 50 | 1347 | 73.4 | 504.35 |
| 60 | 1491 | 88.8 | 429.32 |
| 70 | 1635 | 101.7 | 371.34 |
| 80 | 1779 | 115.3 | 341.85 |
| 90 | 1923 | 128.5 | 315.88 |
| 100 | 2067 | 142.4 | 289.04 |
| 200 | 3507 | 277.6 | 185.34 |
| 300 | 4947 | 415.5 | 151.15 |
| 400 | 6387 | 550.7 | 129.42 |
| 500 | 7827 | 686.7 | 115.50 |
| 600 | 9267 | 825.2 | 106.28 |
| 700 | 10707 | 960.8 | 98.62 |
| 800 | 12147 | 1100.3 | 94.22 |

**Table 4**
Preprocessing time in seconds for real-world graphs.

| Instance | FELINE | BFL | LYNX |
|---|---|---|---|
| arXiv | $< 0.01$ | $< 0.01$ | 17.13 |
| go | $< 0.01$ | $< 0.01$ | 12.24 |
| pubmed | $< 0.01$ | $< 0.01$ | 15.37 |
| citeseer | 0.25 | 0.11 | 36.81 |
| uniprotenc_22m | 0.53 | 0.17 | 66.20 |
| cit-Patents | 4.76 | 3.32 | 150.68 |
| citeseerx | 3.87 | 2.67 | 179.54 |
| go_uniprot | 3.95 | 3.06 | 242.03 |
| govwild | 4.09 | 2.56 | 223.18 |
| uniprotenc_100m | 6.71 | 2.17 | 537.50 |
| yago | 10.01 | 4.39 | 519.93 |
| twitter | 5.69 | 2.35 | 493.08 |
| web-uk | 7.55 | 3.14 | 638.39 |
| uniprotenc_150m | 11.63 | 3.75 | 859.99 |

### 6.5. Results for real-world graphs

In our experiments on real-world graphs, each graph is tested in ten repetitions with random queries and ten repetitions with balanced queries, each repetition using an independent set of one million queries. The results reported here are average values over the relevant repetitions, unless otherwise stated. Table 4 shows preprocessing time of FELINE, BFL and LYNX for real-world benchmark instances. Note that LYNX needs more time than FELINE and BFL to create its index due to the larger size of the index in comparison with FELINE, and more sophisticated algorithms involved. However, the time spent is still reasonable considering graphs with millions of vertices and arcs that will be preprocessed once and queried many times.

The memory usage for FELINE, BFL and LYNX is shown in Table 5. The index size of FELINE increases as the number of vertices grows, to store the two topological sorts of the negative cut index and the set of interval of the positive cut index. For BFL, the index size increases as the number of vertices and arcs is increased, since we fixed the $s$ parameter.

**Table 5**
Memory usage (MB) for real-world graphs.

| Instance | FELINE | BFL | LYNX |
|---|---|---|---|
| arXiv | 51 | 700 | 710 |
| go | 50 | 700 | 710 |
| pubmed | 51 | 701 | 711 |
| citeseer | 170 | 1010 | 983 |
| uniprotenc_22m | 356 | 1398 | 1395 |
| cit-Patents | 908 | 2549 | 2,554 |
| citeseerx | 1356 | 3714 | 3,748 |
| go_uniprot | 1552 | 4090 | 4,154 |
| govwild | 1,742 | 4,474 | 4,448 |
| uniprotenc_100m | 3127 | 7797 | 7694 |
| yago | 3225 | 8052 | 8031 |
| twitter | 3428 | 8615 | 8567 |
| web-uk | 4447 | 10,918 | 11,105 |
| uniprotenc_150m | 4849 | 11,777 | 11,691 |

**Table 6**
Query time (ms) for 1 million random queries on real-world benchmark instances.

| Instance | R-ratio | FELINE | BFL | LYNX | $\frac{LYNX}{BFL}$ | 95% C.I. |
|---|---|---|---|---|---|---|
| arXiv | 15.52 % | 767.2 | 228.6 | 108.5 | 0.475 | [0.470, 0.479] |
| go | 0.24 % | 105.7 | 42.3 | 26.4 | 0.623 | [0.605, 0.641] |
| pubmed | 0.64 % | 82.6 | 41.1 | 28.5 | 0.694 | [0.682, 0.705] |
| citeseer | 0.00 % | 65.3 | 27.3 | 18.7 | 0.686 | [0.672, 0.700] |
| uniprotenc_22m | 0.00 % | 32.4 | 25.2 | 16.4 | 0.652 | [0.639, 0.665] |
| cit-Patents | 0.04 % | 5514.9 | 82.3 | 440.4 | 5.350 | [5.267, 5.432] |
| citeseerx | 0.23 % | 165.5 | 35.7 | 45.6 | 1.277 | [1.250, 1.305] |
| go_uniprot | 0.00 % | 42.2 | 24.2 | 14.8 | 0.609 | [0.599, 0.620] |
| govwild | 0.01 % | 145.1 | 32.7 | 45.6 | 1.395 | [1.358, 1.432] |
| uniprotenc_100m | 0.00 % | 48.6 | 28.6 | 22.0 | 0.770 | [0.752, 0.787] |
| yago | 0.00 % | 54.1 | 27.4 | 25.7 | 0.938 | [0.931, 0.945] |
| twitter | 7.38 % | 97.1 | 34.5 | 40.4 | 1.172 | [1.155, 1.189] |
| web-uk | 8.98 % | 246.1 | 61.1 | 53.9 | 0.883 | [0.863, 0.904] |
| uniprotenc_150m | 0.00 % | 51.6 | 30.6 | 24.9 | 0.813 | [0.803, 0.824] |
| geom. mean | | | | | 0.923 | |

**Table 7**
Query time (ms) for 1 million balanced queries on real-world benchmark instances.

| Instance | FELINE | BFL | LYNX | $\frac{LYNX}{BFL}$ | 95% C.I. |
|---|---|---|---|---|---|
| arXiv | 991.3 | 367.1 | 163.2 | 0.444 | [0.438, 0.451] |
| go | 100.9 | 73.9 | 43.3 | 0.586 | [0.579, 0.593] |
| pubmed | 183.6 | 120.4 | 74.7 | 0.621 | [0.612, 0.630] |
| citeseer | 89.7 | 69.5 | 44.4 | 0.639 | [0.629, 0.650] |
| uniprotenc_22m | 86.5 | 57.6 | 33.7 | 0.584 | [0.580, 0.589] |
| cit-Patents | 18605.6 | 870.3 | 2186.5 | 2.513 | [2.487, 2.538] |
| citeseerx | 304.3 | 125.9 | 125.0 | 0.996 | [0.954, 1.038] |
| go_uniprot | 390.4 | 401.2 | 137.1 | 0.342 | [0.339, 0.344] |
| govwild | 275.8 | 108.9 | 115.4 | 1.060 | [1.040, 1.079] |
| uniprotenc_100m | 141.1 | 83.1 | 56.0 | 0.673 | [0.665, 0.681] |
| yago | 67.6 | 39.5 | 48.5 | 1.228 | [1.217, 1.239] |
| twitter | 62.3 | 29.4 | 33.3 | 1.133 | [1.117, 1.148] |
| web-uk | 200.0 | 98.4 | 73.6 | 0.747 | [0.735, 0.760] |
| uniprotenc_150m | 164.7 | 98.4 | 68.7 | 0.698 | [0.692, 0.704] |
| geom. mean | | | | 0.772 | |

The memory usage of LYNX is limited to the one used by BFL for all instances tested with possibly a small variation due to how operational system allocates memory.

Table 6 shows the average query time for 10 executions of one million random queries. A random query is created by picking random initial and final vertices. BFL and LYNX outperform FELINE for all instances tested. The results obtained by LYNX are on average slightly better than those obtained by BFL. However, BFL achieved shorter query times for 4 out of 14 instances. It is important to note that most instances have a small average vertex degree and also the number of positive random queries (R-ratio) is less than 1 %. Although this instance set is commonly used on literature, these numbers show that they have very particular features that can influence the comparison with approaches that are focused on graphs with lower R-ratio, like BFL.

The table also shows 95% confidence intervals for the value of $\frac{LYNX}{BFL}$, which measures the relative performance of LYNX compared to BFL. Since each execution uses one million queries, the standard deviation for the performance metric is small, and the confidence interval is very narrow. A paired t-test is used to compare the average query time of LYNX and BFL over the ten executions, revealing that the differences are statistically significant at any reasonable level of confidence.

Next, Table 7 shows average results for 10 executions of 500 thousand negative and 500 thousand positive queries. With this test we intend to analyze the performance of LYNX in comparison to BFL and FELINE in a possible realistic scenario with a larger set of positive queries. Again, LYNX has better results for 10 out of 14 instances, with a rather large average improvement over BFL. In this case, paired t-tests reveal that the differences are statistically significant for all instances except the citeseerx instance, where the P-value is 0.355. Also note that for this instance the 95% confidence interval for $\frac{LYNX}{BFL}$ contains the value 1.

### 6.6. Results for synthetic graphs

Table 8 shows the index preprocessing time for synthetic instances with 10 million vertices and average vertex outdegree between 1 to 20. The preprocessing time of LYNX is higher than FELINE and BFL. However, it grows slowly as the number of arcs is increased. In fact, while BFL increases its preprocessing time from 3.22 s for the smallest instance to 25.54 s for the largest (an increase of 7.9 times), LYNX preprocessing time increases from 314.0 to 904.65 s for the same instances (an increase of 2.9 times).

The memory usage for synthetic instances is shown in Table 9. Again, FELINE uses less memory than BFL and LYNX. The memory needed by BFL increases as the number of arcs increase for a fixed *s*. The memory usage for LYNX is limited by the one used for BFL for each instance.

**Table 8**
Preprocessing time (s) for synthetic instances.

| Instance | FELINE | BFL | LYNX |
|---|---|---|---|
| syn-10M-1 | 7.58 | 3.22 | 314.00 |
| syn-10M-2 | 10.05 | 4.91 | 376.19 |
| syn-10M-3 | 11.23 | 6.27 | 405.51 |
| syn-10M-4 | 12.11 | 7.61 | 426.92 |
| syn-10M-5 | 12.95 | 8.93 | 447.52 |
| syn-10M-6 | 13.72 | 10.17 | 472.29 |
| syn-10M-7 | 14.43 | 11.43 | 492.01 |
| syn-10M-8 | 15.11 | 12.39 | 516.34 |
| syn-10M-9 | 15.71 | 13.66 | 539.03 |
| syn-10M-10 | 16.49 | 14.80 | 566.45 |
| syn-10M-11 | 17.35 | 16.24 | 606.11 |
| syn-10M-12 | 18.06 | 17.03 | 637.76 |
| syn-10M-13 | 18.58 | 18.10 | 669.48 |
| syn-10M-14 | 19.40 | 19.20 | 708.53 |
| syn-10M-15 | 20.25 | 20.29 | 731.35 |
| syn-10M-16 | 20.70 | 21.33 | 762.31 |
| syn-10M-17 | 21.32 | 22.65 | 817.96 |
| syn-10M-18 | 22.00 | 23.40 | 846.62 |
| syn-10M-19 | 22.68 | 24.55 | 881.66 |
| syn-10M-20 | 23.45 | 25.54 | 904.65 |

Table 10 shows the query time of for 1 million random queries. The query time of FELINE grows faster than BFL and LYNX, which can be also observed graphically on Fig. 4. For an average vertex degree up to 11, with a small R-ratio (number of positive queries), BFL overcomes LYNX. However, as the number of arcs grows and also the R-ratio, LYNX clearly outperforms BFL and the difference between the two becomes larger.

For benchmark instance `syn-10-20`, with an R-ratio of 22.97 %, LYNX runs in less then 40 % of the time used by BFL. For this instance, LYNX uses 904 s to generate its index and BFL uses 25 s. The preprocessing time and query time of 1 million queries together is 1,760 s for LYNX against 2,210 for BFL. Therefore, even though the preprocessing time for LYNX is much higher than BFL, LYNX overcomes BFL by far in the overall result for instances with high R-ratio.

From these experiments we can conclude that LYNX succeeded in improving the performance of FELINE. The improvement is such, that LYNX is competitive with the state of the art algorithm. There is not a clear dominating approach between LYNX and BFL. LYNX appear to be better on average in the benchmark data-set. Besides, on synthetic instances with 10 million vertices BFL tends to be better only for very sparse graphs while the performance of LYNX is much better than BFL for denser graphs.

When considering algorithms for connectivity queries in graphs, one may wish to find an algorithm that both minimizes query times and

**Table 9**
Memory usage (MB) for synthetic instances.

| Instance | FELINE | BFL | LYNX |
|---|---|---|---|
| syn-10M-1 | 1947 | 5090 | 4815 |
| syn-10M-2 | 2095 | 5240 | 5031 |
| syn-10M-3 | 2207 | 5372 | 5258 |
| syn-10M-4 | 2311 | 5522 | 5415 |
| syn-10M-5 | 2417 | 5683 | 5583 |
| syn-10M-6 | 2526 | 5852 | 5827 |
| syn-10M-7 | 2635 | 6027 | 5994 |
| syn-10M-8 | 2753 | 6208 | 6241 |
| syn-10M-9 | 2878 | 6392 | 6422 |
| syn-10M-10 | 2994 | 6577 | 6604 |
| syn-10M-11 | 3114 | 6760 | 6859 |
| syn-10M-12 | 3224 | 6936 | 7107 |
| syn-10M-13 | 3328 | 7108 | 7275 |
| syn-10M-14 | 3441 | 7284 | 7519 |
| syn-10M-15 | 3559 | 7463 | 7687 |
| syn-10M-16 | 3685 | 7649 | 7811 |
| syn-10M-17 | 3813 | 7843 | 8127 |
| syn-10M-18 | 3948 | 8043 | 8321 |
| syn-10M-19 | 4083 | 8241 | 8590 |
| syn-10M-20 | 4211 | 8440 | 8703 |

**Table 10**
Query time (ms) for 1 million random queries on synthetic benchmark instances

| Instance | R-ratio | FELINE | BFL | LYNX | $\frac{LYNX}{BFL}$ |
|---|---|---|---|---|---|
| syn-10M-1 | 0.00 % | 80.07 | 34.27 | 37.04 | 108.07 % |
| syn-10M-2 | 0.00 % | 283.27 | 44.45 | 52.57 | 118.26 % |
| syn-10M-3 | 0.00 % | 990.78 | 47.28 | 110.10 | 232.86 % |
| syn-10M-4 | 0.00 % | 4,143.29 | 60.28 | 359.48 | 596.39 % |
| syn-10M-5 | 0.01 % | 19,189.83 | 223.53 | 1,513.31 | 677.00 % |
| syn-10M-6 | 0.12 % | 88,920.91 | 1,480.11 | 6,729.79 | 454.68 % |
| syn-10M-7 | 0.57 % | 371,138.62 | 9,023.79 | 28,471.64 | 315.52 % |
| syn-10M-8 | 1.66 % | 1,011,163.88 | 33,194.64 | 78,998.78 | 237.99 % |
| syn-10M-9 | 3.47 % | 2,039,423.75 | 86,928.96 | 164,553.57 | 189.30 % |
| syn-10M-10 | 5.56 % | 3,094,120.25 | 171,151.09 | 255,431.69 | 149.24 % |
| syn-10M-11 | 7.80 % | 4,143,705.25 | 284,854.66 | 346,045.91 | 121.48 % |
| syn-10M-12 | 9.97 % | 5,044,193.00 | 437,610.60 | 432,520.21 | 98.84 % |
| syn-10M-13 | 12.05 % | 5,816,364.50 | 607,268.01 | 508,844.34 | 83.79 % |
| syn-10M-14 | 14.00 % | 6,793,176.50 | 811,775.72 | 562,137.75 | 69.25 % |
| syn-10M-15 | 15.80 % | 7,713,948.50 | 1,019,217.07 | 616,853.78 | 60.52 % |
| syn-10M-16 | 17.43 % | 8,062,516.00 | 1,235,536.72 | 661,500.98 | 53.54 % |
| syn-10M-17 | 19.06 % | 8,315,207.00 | 1,478,750.13 | 731,334.65 | 49.46 % |
| syn-10M-18 | 20.39 % | 9,561,146.00 | 1,698,724.03 | 772,635.99 | 45.48 % |
| syn-10M-19 | 21.67 % | 9,918,577.00 | 1,925,852.27 | 772,956.71 | 40.14 % |
| syn-10M-20 | 22.97 % | 9,804,729.00 | 2,185,029.81 | 856,852.79 | 39.21 % |

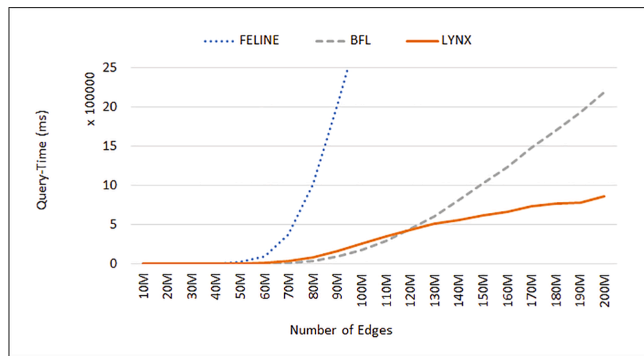



**Fig. 4.** FELINE, BFL and LYNX comparison on query time for random queries as the number of arcs is increased.

preprocessing times. As illustrated above, this instead forms a trade-off, where algorithms that invest more time in the preprocessing phase can gain an advantage in the phase of actively testing queries. LYNX is not dominated by any existing algorithms in terms of this trade-off, as no algorithm currently has both lower preprocessing times and query times than LYNX.

Both LYNX and BFL could be further improved by using the SCARAB framework (Jin et al., 2012) and DAG Reduction (Zhou et al., 2017; Zhou et al., 2018) which can speed up *Refined Online Search* approaches by reducing the graph size. Such experiment is outside the scope of this work.

## 7. Conclusion

In this work, we showed opportunities of improvement over recent approaches for reachability on very large graphs. A pathological case for FELINE is identified. We showed that the heuristic used by FELINE solves a problem whose optimal solution is limited to the eccentricity of the vertex representing $t_X$ in the graph of topological sorts. BFL and IP have a limited use of positive cuts, using the same strategy by Yildirim et al. (2012) that is improved in this work.

We proposed a novel approach called LYNX that overcomes some of the FELINE shortcomings and creates better indices. It allows a larger negative cut and positive cut index increasing its performance.

Computational results show that LYNX outperforms FELINE and has competitive results in comparison with BFL for real-world instances, 9% better in average using the same index size. We show experiments with synthetic instances in which, even using a larger preprocessing time, LYNX consistently outperforms BFL in the overall results (preprocessing plus query time) as the number of arcs and the reachability ratio grows.

From these experiments we can conclude that LYNX succeeded in improving the performance of FELINE. Moreover, the improvement is such, that LYNX is competitive with state-of-the-art algorithms.

However, the main limitation of LYNX lies in its higher preprocessing time. For this reason, it is most suitable for large static networks in which a high number of queries need to be answered fast. Whenever the underlying graph is updated frequently, the added preprocessing time may be prohibitive, and other methods with longer query times may be preferred. This is a trade-off that must be addressed for each application separately, depending on the needs of the end users.

Future work includes collecting more real-world instances with a high number of vertices and arcs, and combining LYNX with the SACARAB framework (Jin et al., 2012) which can speed up *Refined Online Search* approaches by reducing the graph size. Once combined we will be able to compare the results of LYNX with the other state-of-the-art approaches in reduced graphs. Finally, rather than working on graphs extracted from various applications, it remains to implement LYNX as a part of an actual expert system to evaluate how the improved query times can influence the behavior of the end users.

## CRediT authorship contribution statement

**Rodrigo Ferreira da Silva:** Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Writing - original draft, Writing - review & editing. **Sebastián Urrutia:** Conceptualization, Formal analysis, Investigation, Methodology, Project administration, Supervision, Validation, Writing - original draft, Writing - review & editing. **Lars Magnus Hvattum:** Formal analysis, Investigation, Methodology, Project administration, Supervision, Writing - review & editing.

## Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence

the work reported in this paper.

## Acknowledgments

Rodrigo Ferreira da Silva thanks CAPES for his PhD grant. Sebastián Urrutia was partially supported by CNPq research grant 304475/2016–5. This work was partially supported by the Norwegian Centre for International Cooperation in Education (SIU) project number UTF-2016-short-term/10123.

## References

Anand, A., Seufert, S., Bedathur, S., & Weikum, G. (2013). FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *Proceedings of the 2013 ieee international conference on data engineering (icde 2013)* (pp. 1009–1020). Washington, DC, USA: IEEE Computer Society.

Brightwell, G. R., & Massow, M. (2013). Diametral pairs of linear extensions. *SIAM Journal on Discrete Mathematics, 27*(2), 634–649. https://doi.org/10.1137/080733140

Broder, A. (1997). On the resemblance and containment of documents. In *Proceedings of the compression and complexity of sequences 1997* (p. 21). Washington, DC, USA: IEEE Computer Society.

Cha, M., Haddadi, H., Benevenuto, F., & Gummadi, K. P. (2010). Measuring user influence in twitter: The million follower fallacy. In *icwsm'10: Proceedings of international aaai conference on weblogs and social*.

Chen, Y. (2009). General spanning trees and reachability query evaluation. In *Proceedings of the 2nd canadian conference on computer science and software engineering* (pp. 243–252). New York, NY, USA: ACM.

Cheng, J., Huang, S., Wu, H., & Fu, A. W.-C. (2013). Tf-label: A topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of the 2013 acm sigmod international conference on management of data* (pp. 193–204). New York, NY, USA: ACM.

Corneil, D. G., Dragan, F. F., & Köhler, E. (2002). On the power of bfs to determine a graphs diameter. In S. Rajsbaum (Ed.), *Latin 2002: Theoretical informatics* (pp. 209–223). Berlin, Heidelberg: Springer, Berlin Heidelberg.

de Silva, R. F., Urrutia, S., & dos Santos, V. (2019). One-sided weak dominance drawing. *Theoretical Computer Science, 757*, 36–43.

He, H., Wang, H., Yang, J., & Yu, P. S. (2005). Compact reachability labeling for graph-structured data. In *Proceedings of the 14th acm international conference on information and knowledge management* (pp. 594–601). New York, NY, USA: ACM.

Jin, R., Ruan, N., Dey, S., & Xu, J. Y. (2012). SCARAB: Scaling reachability computation on large graphs. In *Proceedings of the 2012 acm sigmod international conference on management of data* (pp. 169–180). New York, NY, USA: ACM.

Jin, R., Ruan, N., Xiang, Y., & Wang, H. (2011). Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Transactions on Database Systems, 36* (1), 7:1–7:44.

Jin, R., & Wang, G. (2013, September). Simple, fast, and scalable reachability oracle. Proc. VLDB Endow., 6 (14), 1978–1989.

Jin, R., Xiang, Y., Ruan, N., & Fuhry, D. (2009). 3hopp: A high-compression indexing scheme for reachability query. In *Proceedings of the 2009 acm sigmod international conference on management of data* (pp. 813–826). New York, NY, USA: ACM.

Kleinberg, J. M., Kumar, R., Raghavan, P., Rajagopalan, S., & Tomkins, A. S. (1999). The web as a graph: Measurements, models, and methods. In *Proceedings of the 5th annual international conference on computing and combinatorics* (pp. 1–17). Berlin, Heidelberg: Springer-Verlag.

Kornaropoulos, E. M. (2012). *Dominance Drawing of Non-Planar Graphs (Unpublished master's thesis)*. Heraklion, Greece: University of Crete.

Kornaropoulos, E.M., & Tollis, I.G. (2011). Weak dominance drawings and linear extension diameter. CoRR, abs/1108.1439.

Kornaropoulos, E.M., & Tollis, I.G. (2012). Weak dominance drawings for directed acyclic graphs. In Graph drawing - 20th international symposium, GD 2012, redmond, wa, usa, september 19–21, 2012, revised selected papers (pp. 559–560).

Li, L., Hua, W., & Zhou, X. (2017, Jul 01). HD-GDD: high dimensional graph dominance drawing approach for reachability query. World Wide Web, 20 (4), 677–696.

Massow, M. (2009). *Linear extension graphs and linear extension diameter (Unpublished doctoral dissertation)*. Berlin: TU Berlin.

Su, J., Zhu, Q., Wei, H., & Yu, J. X. (2017). Reachability querying: Can it be even faster? *IEEE Transactions on Knowledge and Data Engineering, 29*(3), 683–697.

Tarjan, R. (1972). Depth first search and linear graph algorithms. *SIAM Journal On Computing, 1*(2).

Veloso, R.R., Cerf, L., Jr., W.M., & Zaki, M.J. (2014). Reachability queries in very large graphs: A fast refined online search approach. In Proceedings of the 17th international conference on extending database technology, EDBT 2014, athens, greece, march 24-28, 2014. (pp. 511–522).

Wang, H., He, H., Yang, J., Yu, P., & Yu, J. (2006). Dual labeling: Answering graph reachability queries in constant time. In 22nd International Conference on Data Engineering (ICDE'06) (p. 75–75).

Wei, H., Yu, J.X., Lu, C., & Jin, R. (2018, 2). Reachability querying: An independent permutation labeling approach. The VLDB Journal, 27 (1), 1–26.

Yildirim, H., Chaoji, V., & Zaki, M.J. (2012, August). GRAIL: A scalable index for reachability queries in very large graphs. The VLDB Journal, 21 (4), 509–534.

Zhou, J., Yu, J. X., Li, N., Wei, H., Chen, Z., & Tang, X. (2018). Accelerating reachability query processing based on DAG reduction. *The VLDB Journal, 27*(2), 271–296.

Zhou, J., Zhou, S., Yu, J. X., Wei, H., Chen, Z., & Tang, X. (2017). DAG reduction: Fast answering reachability queries. In *Proceedings of the 2017 acm international conference on management of data* (pp. 375–390). New York, NY, USA: ACM.