Faster Fully Dynamic Transitive Closure in Practice

Kathrin Hanauer
University of Vienna, Faculty of Computer Science, Austria
kathrin.hanauer@univie.ac.at

Monika Henzinger
University of Vienna, Faculty of Computer Science, Austria
monika.henzinger@univie.ac.at

Christian Schulz
University of Vienna, Faculty of Computer Science, Austria
christian.schulz@univie.ac.at

Abstract

The fully dynamic transitive closure problem asks to maintain reachability information in a directed graph between arbitrary pairs of vertices, while the graph undergoes a sequence of edge insertions and deletions. The problem has been thoroughly investigated in theory and many specialized algorithms for solving it have been proposed in the last decades. In two large studies [Frigioni ea, 2001; Krommidas and Zaroliagis, 2008], a number of these algorithms have been evaluated experimentally against simple, static algorithms for graph traversal, showing the competitiveness and even superiority of the simple algorithms in practice, except for very dense random graphs or very high ratios of queries. A major drawback of those studies is that only small and mostly randomly generated graphs are considered.

In this paper, we engineer new algorithms to maintain all-pairs reachability information which are simple and space-efficient. Moreover, we perform an extensive experimental evaluation on both generated and real-world instances that are several orders of magnitude larger than those in the previous studies. Our results indicate that our new algorithms outperform all state-of-the-art algorithms on all types of input considerably in practice.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Dynamic Graph Algorithms, Reachability, Transitive Closure

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.14

Related Version A full version of the paper is available at [12], `https://arxiv.org/abs/2002.00813`.

Supplementary Material Source code and instances are available at `https://dyreach.taa.univie.ac.at/transitive-closure`.

Funding The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

1 Introduction

Complex graphs are useful in a wide range of applications from technological networks to biological systems like the human brain. These graphs can contain billions of vertices and edges. Analyzing these networks aids us in gaining new insights about our surroundings. One of the most basic questions that arises in this setting is whether one vertex can *reach* another vertex via a directed path. This simple problem has a wide range of applications such program analysis [28], protein-protein interaction networks [10], centrality measures, and is used as subproblem in a wide range of more complex (dynamic) algorithms such as in the computation of (dynamic) maximum flows [8, 6, 11]. Often, the underlying graphs or input instances change over time, i.e., vertices or edges are inserted or deleted as time is passing. In a social network, for example, users sign up or leave, and relations between them may be created or removed over time. Terminology-wise, a problem is said to be *fully dynamic* if the update operations include both insertions *and* deletions of edges, and *partially dynamic* if only one type of update operations is allowed. In this context, a problem is called *incremental*, if only edge insertions occur, but no deletions, and *decremental* vice versa.

Recently, we studied an extensive set of algorithms for the *single-source* reachability problem in the fully dynamic setting [13]. The fully dynamic single-source reachability problem maintains the set of vertices that are reachable from a given *source vertex*, subject to edge deletions and insertions. In particular, we designed several fully dynamic variants of well-known approaches to obtain and maintain reachability information with respect to a distinguished source.

This yields the *starting point of this paper*: our goal was to transfer recently engineered algorithms for the fully dynamic *single-source* reachability problem [13] to the more general fully dynamic *transitive closure* problem (also known as fully dynamic all-pairs reachability problem). In contrast to the single-source problem, the *fully dynamic transitive closure* problem consists in maintaining reachability information between *arbitrary* pairs of vertices s and t in a directed graph, which in turn is subject to edge insertions and deletions. If the graph does not change, i.e., in the *static* setting, the question whether an arbitrary vertex s can reach another arbitrary vertex t can either be answered in linear time by starting a breadth-first or depth-first search from s, or it can be answered in constant time after the transitive closure of the graph, i.e., reachability information for all pairs of vertices, has been computed. The latter can be obtained in $\mathcal{O}(n^{\omega})$, where ω is the exponent in the running time of the best known fast matrix multiplication algorithm (currently, $\omega < 2.38$ [24]), or combinatorially in $\mathcal{O}(n \cdot m)$ or $\mathcal{O}(n^3)$ time by either starting a breadth-first or depth-first search from each vertex or using the Floyd-Warshall algorithm [7, 38, 3].

In the dynamic setting, the aim is to avoid such costly recomputations from scratch after the graph has changed, especially if the update was small. Hence, the dynamic version of the problem has been thoroughly studied in theory and many specialized algorithms for solving it have been proposed in the last decades. However, even the insertion or deletion of a single edge may affect the reachability between $\Omega(n^2)$ vertex pairs, which is why one cannot hope for an algorithm with constant query time that processes updates in less than $\mathcal{O}(n^2)$ worst-case time if the transitive closure is maintained explicitly. Furthermore, conditional lower bounds [15] suggest that no faster solution than the naïve recomputation from scratch is possible after each change in the graph.

Whereas the static approach to compute the transitive closure beforehand via graph traversal can be readily adapted to the incremental setting, yielding an amortized update time of $\mathcal{O}(n)$ [17], a large number of randomized and deterministic algorithms [16, 18, 14, 20, 19, 21, 4, 5, 31, 29, 34, 27] has been devised over the last years for the decremental and the fully dynamic version of the problem. The currently fastest algorithm in the deletions-only case is deterministic, has a total update time of $\mathcal{O}(n \cdot m)$, and answers queries in constant time [27]. In the fully dynamic setting, updates can be processed deterministically in $\mathcal{O}(n^2)$ amortized time with constant query time [5], or, alternatively in $\mathcal{O}(m\sqrt{n})$ amortized update time with $\mathcal{O}(\sqrt{n})$ query time [31]. An almost exhaustive set of these algorithms has been evaluated experimentally against simple, static algorithms for graph traversal such as breadth-first or depth-first search in two large studies [9, 22]. Surprisingly, both have shown that the simple algorithms are competitive and even superior to the specialized algorithms in practice, except for dense random graphs or, naturally, very high ratios of queries. Only two genuinely dynamic algorithms could challenge the simple ones: an algorithm developed by Frigioni et al. [9], which is based on Italiano's algorithms [17, 18] as well as an extension of a decremental Las Vegas algorithm proposed by Roditty and Zwick [31], developed by Krommidas and Zaroliagis [22]. Both rely on the computation and maintenance of strongly connected components, which evidently gives them an advantage on dense graphs. Nevertheless, they appeared to be unable to achieve a speedup of a factor greater than ten in comparison to breadth-first and depth-first search.

In this paper, we engineer a family of algorithms that build on recent experimental results for the single-source reachability problem [13]. Our algorithms are very easy to implement and benefit from strongly connected components likewise, although they do not (necessarily) compute them explicitly. In an extensive experimental evaluation on various types of input instances, we compare our algorithms to all simple algorithms from [9, 22] as well as a modified, bidirectional breadth-first search. The latter already achieves a speedup of multiple factors over the standard version, and our new algorithms outperform the simple algorithms on all types of input by several orders of magnitude in practice.

2 Preliminaries

Basic Concepts. Let $G = (V, E)$ be a directed graph with vertex set V and edge set E. Throughout this paper, let $n = |V|$ and $m = |E|$. The *density* of G is $d = \frac{m}{n}$. An edge $(u, v) \in E$ has *tail* u and *head* v and u and v are said to be *adjacent*. (u, v) is said to be an *outgoing* edge or *out-edge* of u and an *incoming* edge or *in-edge* of v. The *outdegree* $\deg^+(v)$/*indegree* $\deg^-(v)$/*degree* $\deg(v)$ of a vertex v is its number of (out-/in-) edges. The *out-neighborhood* (*in-neighborhood*) of a vertex u is the set of all vertices v such that $(u, v) \in E$ ($(v, u) \in E$). A sequence of vertices $s \to \cdots \to t$ such that each pair of consecutive vertices is connected by an edge is called an *s-t path* and s can *reach* t. A *strongly connected component* (*SCC*) is a maximal subset of vertices $X \subseteq V$ such that for each ordered pair of vertices $s, t \in X$, s can reach t. The *condensation* of a directed graph G is a directed graph G_C that is obtained by shrinking every SCC of G to a single vertex, thereby preserving edges between different SCCs. A graph is *strongly connected* if it has only one SCC. In case that each SCC is a singleton, i.e., G has n SCCs, G is said to be *acyclic* and also called a *DAG* (directed acyclic graph). The *reverse* of a graph G is a graph with the same vertex set as G, but contains for each edge $(u, v) \in E$ the reverse edge (v, u).

A *dynamic graph* is a directed graph G along with an ordered sequence of updates, which consist of edge insertions and deletions. In this paper, we consider the *fully dynamic transitive closure problem*: Given a directed graph, answer reachability queries between arbitrary pairs of vertices s and t, subject to edge insertions and deletions.

Related Work. Due to space limitations, we only give a brief overview over related work by listing the currently best known results for fully dynamic algorithms on general graphs in Table 1. For details and partially dynamic algorithms, see the full version [12].

In large studies, Frigioni et al. [9] as well as Krommidas and Zaroliagis [22] have implemented an extensive set of known algorithms for dynamic transitive closure and compared them to each other as well as to simple, static algorithms such as breadth-first and depth-first search. The set comprises the original algorithms in [17, 18, 39, 14, 19, 21, 29, 31, 33, 4, 5] as well as several modifications and improvements thereof. The experimental evaluations on random Erdős-Renyí graphs, instances constructed to be difficult on purpose, as well

Table 1 Currently best results for fully dynamic transitive closure. All running times are asymptotic ($\mathcal{O}$-notation).

Query Time	Update Time	
m	1	naïve
1	n^2	Demetrescu and Italiano [5], Roditty [29], Sankowski [34]
$\sqrt{n}$	$m\sqrt{n}$	Roditty and Zwick [31]
$m^{0.43}$	$m^{0.58}n$	Roditty and Zwick [31]
$n^{0.58}$	$n^{1.58}$,	Sankowski [34]
$n^{1.495}$	$n^{1.495}$	Sankowski [34]
n	$m + n \log n$	Roditty and Zwick [33]
$n^{1.407}$	$n^{1.407}$	van den Brand et al. [37]

as two instances based on real-world graphs, showed that on all instances except for dense random graphs or a query ratio of more than 65 %, the simple algorithms outperformed the dynamic ones distinctly and up to several factors. Their strongest competitors were the fully dynamic extension [9] of the algorithms by Italiano [17, 18], as well as the fully dynamic extension [22] of the decremental algorithm by Roditty and Zwick [31]. These two algorithms also were the only ones that were faster than static graph traversal on dense random graphs, by a factor of at most ten.

3 Algorithms

We propose a new and very simple approach to maintain the transitive closure in a fully dynamic setting. Inspired by a recent study on single-source reachability, it is based solely on single-source and single-sink reachability (SSR) information. Unlike most algorithms for dynamic transitive closure, it does not explicitly need to compute or maintain strongly connected components – which can be time-consuming – but, nevertheless, profits indirectly if the graph is strongly connected. Different variants and parameterizations of this approach lead to a family of new algorithms, all of which are easy to implement and – depending on the choice of parameters – extremely space-efficient.

In Section 4, we evaluate this approach experimentally against a set of algorithms that have been shown to be among the fastest algorithms in practice so far. This set includes the classic simple, static algorithms for graph traversal, breadth-first search and depth-first search. For the sake of completeness, we will start by describing the practical state-of-the-art algorithms, and then continue with our new approach. Each algorithm for fully dynamic transitive closure can be described by means of four subroutines: $\mathsf{initialize}()$, $\mathsf{insertEdge}((u, v))$, $\mathsf{deleteEdge}((u, v))$, and $\mathsf{query}(s, t)$, which define the behavior during the initialization phase, in case that an edge (u, v) is added or removed, and how it answers a query of whether a vertex s can reach a vertex t, respectively.

Table 2 provides an overview of all algorithms in this section along with their abbreviations. All algorithms considered are combinatorial and either deterministic or Las Vegas-style randomized, i.e., their running time, but not their correctness, may depend on random variables.

3.1 Static Algorithms

In the static setting or in case that a graph is given without further (reachability) information besides its edges, breadth-first search (*BFS*) and depth-first search (*DFS*) are the two standard algorithms to determine whether there is a path between a pair of vertices or not. Despite their simplicity and the fact that they typically have no persistent memory (such as a cache, e.g.), experimental studies [9, 22] have shown them to be at least competitive with both partially and fully dynamic algorithms and even superior on various instances.

BFS, DFS. We consider both BFS and DFS in their pure versions: For each query(s, t), a new BFS or DFS, respectively, is initiated from s until either t is encountered or the graph is exhausted. The algorithms store or maintain no reachability information whatsoever and do not perform any work in initialize(), insertEdge$((u, v))$, or deleteEdge$((u, v))$. We refer to these algorithms simply as `BFS` and `DFS`, respectively.

In addition, we consider a hybridization of `BFS` and `DFS`, called `DBFS`, which was introduced originally by Frigioni et al. [9] and is also part of the later study [22]. In case of a query(s, t), the algorithm visits vertices in DFS order, starting from s, but additionally checks for each vertex that is encountered whether t is in its out-neighborhood.

Bidirectional BFS. To speed up static reachability queries even further, we adapted a well-established approach for the more general problem of finding shortest paths and perform two breadth-first searches alternatingly: Upon a query(s, t), the algorithm initiates a customary BFS starting from s, but pauses already after few steps, even if t has not been encountered yet. The algorithm then initiates a BFS on the reverse graph, starting from t, and also pauses after few steps, even if s has not been encountered yet. Afterwards, the first and the second BFS are resumed by turns, always for a few steps only, until either one of them encounters a vertex v that has already encountered by the other, or the graph is exhausted. In the former case, there is a path from s via v to t, hence the algorithm answers the query positively, and otherwise negatively. We refer to this algorithm as `BiBFS` (*Bidirectional BFS*) and use the respective out-degree of the current vertex in each BFS as step size, i.e., each BFS processes one vertex, examines all its out-neighbors, and then pauses execution. Note that the previous experimental studies [9, 22] do not consider this algorithm.

3.2 A New Approach

General Overview. Let v be an arbitrary vertex of the graph and let $R^+(v)$ and $R^-(v)$ be the sets of vertices reachable from v and that can reach v, respectively. To answer reachability queries between two vertices s and t, we use the following simple observations:

(O1) If $s \in R^-(v)$ and $t \in R^+(v)$, then s can reach t.
(O2) If v can reach s, but not t, i.e., $s \in R^+(v)$ and $t \notin R^+(v)$, then s cannot reach t.
(O3) If t can reach v, but s cannot, i.e., $s \notin R^-(v)$ and $t \in R^-(v)$, then s cannot reach t.

Whereas the first observation is widely used in several algorithms, we are not aware of any algorithms making direct use of the others. Our new family of algorithms keeps a list $\mathcal{L}_{SV}$ of length k of so-called *supportive vertices*, which work similarly to cluster centers in decremental shortest paths algorithms [32]. For each vertex v in $\mathcal{L}_{SV}$, there are two fully dynamic data structures maintaining the sets $R^+(v)$ and $R^-(v)$, respectively. In other words, these data structures maintain single-source as well as single-sink reachability for a vertex v. We give details on those data structures at the end of this section. All updates to the graph, i.e., all notifications of insertEdge(·) and deleteEdge(·), are simply passed on to these data structures. In case of a query(s, t), the algorithms first check whether one of s or t is a supportive vertex itself. In this case, the query can be answered decisively using the corresponding data structure. Otherwise, the vertices in $\mathcal{L}_{SV}$ are considered one by one and the algorithms try to apply one of the above observations. Finally, if this also fails, a static algorithm serves as fallback to answer the reachability query.

Whereas this behavior is common to all algorithms of the family, they differ in their choice of supportive vertices and the subalgorithms used to maintain SSR information as well as the static fallback algorithm.

Note that it suffices if an algorithm has exactly one vertex v_i from each SCC C_i in $\mathcal{L}_{SV}$ to answer every reachability query in the same time as a query to a SSR data structure, e.g., $\mathcal{O}(1)$: If s and t belong to the same SCC C_i, then the supportive vertex v_i is reachable from s and can reach t, so the algorithm answers the query positively in accordance with observation **(O1)**. Otherwise, s belongs to an SCC C_i and t belongs to an SCC C_j, $j \neq i$. If C_i can reach C_j in the condensation of the graph, then also v_i can reach t and v_i is reachable from s, so the algorithm again answers the query positively in accordance with observation **(O1)**. If C_i cannot reach C_j in the condensation of the graph, then v_i can reach s, but not t so the algorithm answers the query negatively in accordance with observation **(O2)**. The supportive vertex representing the SCC that contains s or t, respectively, may be found in constant time using a map; however, updating it requires in turn to maintain the SCCs dynamically, which incurs additional costs during edge insertions and deletions.

Choosing Supportive Vertices. The simplest way to choose supportive vertices consists in picking them uniformly at random from the set of all vertices in the initial graph and never revisiting this decision. We refer to this algorithm as SV(k) (*k-Supportive Vertices*). During initialize(), SV(k) creates $\mathcal{L}_{SV}$ by drawing k non-isolated vertices uniformly at random from V. For each $v \in \mathcal{L}_{SV}$, it initializes both a dynamic single-source as well as a dynamic single-sink reachability data structure, each rooted at v. If less than k vertices have been picked during initialization because of isolated vertices, $\mathcal{L}_{SV}$ is extended as soon as possible.

Naturally, the initial choice of supportive vertices may be unlucky, which is why we also consider a variation of the above algorithm that periodically clears the previously chosen list of supportive vertices after c update operations and re-runs the selection process. We refer to this algorithm as SVA(k, c) (*k-Supportive Vertices with c-periodic Adjustments*).

As shown above, the perfect choice of a set of supportive vertices consists of exactly one per SCC. This is implemented by the third variant of our algorithm, SVC (*Supportive Vertices with SCC Cover*). However, maintaining SCCs dynamically is itself a non-trivial task and has recently been subject to extensive research [27, 30]. Here, we resolve this problem by waiving exactness, or, more precisely, the reliability of the cover. Similar to above, the algorithm takes two parameters z and c. During initialize(), it computes the SCCs of the input graph and arbitrarily chooses a supportive vertex in each SCC as representative if the SCC's size is at least z. In case that all SCCs are smaller than z, an arbitrary vertex that is neither a source nor a sink, if existent, is made supportive. The algorithm additionally maps each vertex to the representative of its SCC, where possible. After c update operations, this process is re-run and the list of supportive vertices as well as the vertex-to-representative map is updated suitably. However, we do not de-select supportive vertices picked in a previous round if they represent an SCC of size less than z, which would mean to also destroy their associated SSR data structures. Recall that computing the SCCs of a graph can be accomplished in $\mathcal{O}(n + m)$ time [36]. For a query(s, t), the algorithm looks up the SCC representative of s in its map and checks whether this information, if present, is still up-to-date by querying their associated data structures. In case of success, the algorithm answers the query as described

Table 2 Algorithms and abbreviations overview.

Algorithm	Long name	Algorithm	Long name
`DFS` / `BFS`	static DFS / BFS	`SV`	Supportive Vertices
`DBFS`	static DFS-BFS hybrid	`SVA`	Supportive Vertices with Adjustments
`BiBFS`	static bidirectional BFS	`SVC`	Supportive Vertices with SCC Cover

in the ideal scenario by asking whether the representative of s can reach t. Otherwise, the algorithm analogously tries to use the SCC representative of t. Outdated SCC representative information for s or t is deleted to avoid further unsuccessful checks. In case that neither s nor t have valid SCC representatives, the algorithm falls back to the operation mode of `SV`.

Algorithms for Maintaining Single-Source/Single-Sink Reachability. To access fully dynamic SSR information, we consider the two single-source reachability algorithms that have been shown to perform best in an extensive experimental evaluation on various types of input instances [13]. In the following, we only provide a short description and refer the interested reader to the original paper [13] for details.

The first algorithm, `SI`, is a fully dynamic extension of a simple incremental algorithm and maintains a not necessarily height-minimal reachability tree. It starts initially with a BFS tree, which is also extended using BFS in insertEdge((u, v)) only if v and vertices reachable from v were unreachable before. In case of deleteEdge((u, v)), the algorithm tries to reconstruct the reachability tree, if necessary, by using a combination of backward and forward BFS. If the reconstruction is expected to be costly because more than a configurable ratio ρ of vertices may be affected, the algorithm instead recomputes the reachability tree entirely from scratch using BFS. The algorithm has a worst-case insertion time of $\mathcal{O}(n + m)$, and, unless $\rho = 0$, a worst-case deletion time of $\mathcal{O}(n \cdot m)$.

The second algorithm, `SES`, is a simplified, fully dynamic extension of Even-Shiloach trees [35] and maintains a (height-minimal) BFS tree throughout all updates. Initially, it computes a BFS tree for the input graph. In insertEdge((u, v)), the tree is updated where necessary using a BFS starting from v. To implement deleteEdge((u, v)), the algorithm employs a simplified procedure in comparison to Even-Shiloach trees, where the BFS level of affected vertices increases gradually until the tree has been fully adjusted to the new graph. Again, the algorithm may abort the reconstruction and recompute the BFS tree entirely from scratch if the update cost exceeds configurable thresholds ρ and β. For constant β, the worst-case time per update operation (edge insertion or deletion) is in $\mathcal{O}(n + m)$.

Both algorithms have $\mathcal{O}(n + m)$ initialization time, support reachability queries in $\mathcal{O}(1)$ time, and require $\mathcal{O}(n)$ space. We use the same algorithms to maintain single-sink reachability information by running the single-source reachability algorithms on the reverse graph.

4 Experiments

Setup. For the experimental evaluation of our approach and the effects of its parameters, we implemented[1] it together with all four static approaches mentioned in Section 3 in

[1] Source code and instances are available at `https://dyreach.taa.univie.ac.at/transitive-closure`.

C++17 and compiled the code with GCC 7.4 using full optimization (`-O3 -march=native -mtune=native`). We would have liked to include the two best non-static algorithms from the earlier study [22]; unfortunately, the released source code is based on a proprietary algorithm library. Nevertheless, we are able to compare our new algorithms indirectly to both by relating their performance to DFS and BFS, as happened in the earlier study. All experiments were run sequentially under Ubuntu 18.04 LTS with Linux kernel 4.15 on an Intel Xeon E5-2643 v4 processor clocked at 3.4 GHz, where each experiment had exclusive access to one core and could use solely local memory, i.e., in particular no swapping was allowed.

For each algorithm (variant) and instance, we separately measured the time spent *by the algorithm* on updates as well as on queries. We specifically point out that the measured times exclude the time spent on performing an edge insertion or deletion *by the underlying graph data structure*. This is especially of importance if an algorithm's update time itself is very small, but the graph data structure has to perform non-trivial work. We use the dynamic graph data structure from the open-source library Algora [1], which is able to perform edge insertions and deletions in constant time. Our implementation is such that the algorithms are unable to look ahead in time and have to process each operation individually. To keep the numbers easily readable, we use k and m as abbreviations for $\times 10^3$ and $\times 10^6$, respectively.

Instances. We evaluate the algorithms on a diverse set of random and real-world instances, which have also been used in [13].

ER Instances. The random dynamic instances generated according to the Erdős-Renyí model $G(n, m)$ consist of an initial graph with $n = 100\text{k}$ or $n = 10\text{m}$ vertices and $m = d \cdot n$ edges, where $d \in [1.25 \ldots 50]$. In addition, they contain a random sequence of 100k operations σ consisting of edge insertions, edge deletions, as well as reachability queries: For an insertion or a query, an ordered pair of vertices was chosen uniformly at random from the set of all vertices. Likewise, an edge was chosen uniformly at random from the set of all edges for a deletion. The resulting instances may contain parallel edges as well as loops and each operation is contained in a batch of ten likewise operations.

Kronecker Instances. Our evaluation uses two sets of size 20 each: `kronecker-csize` contains instances with $n \approx 130\text{k}$, whereas those in `kronecker-growing` have $n \approx 30$ initially and grow to $n \approx 130\text{k}$ in the course of updates. As no generator for dynamic stochastic Kronecker graph exists, the instances were obtained by computing the differences in edges in a series of so-called *snapshot graphs*, where the edge insertions and deletions between two subsequent snapshot graphs were shuffled randomly. The snapshot graphs where generated by the `krongen` tool that is part of the SNAP software library [26], using the estimated initiator matrices given in [25] that correspond to real-world networks. The instances in `kronecker-csize` originate from ten snapshot graphs with 17 iterations each, which results in update sequences between 1.6m and 702m. As they are constant in size, there are roughly equally many insertions and deletions. Their densities vary between 0.7 and 16.4. The instances in `kronecker-growing` were created from thirteen snapshot graphs with five up to 17 iterations, resulting in 282k to 82m update operations, 66 % to 75 % of which are insertions. Their densities are between 0.9 and 16.4.

Real-World Instances. Our set of instances comprises all six directed, dynamic instances available from the Koblenz Network Collection KONECT [23], which correspond to the hyperlink network of Wikipedia articles for six different languages. In case of dynamic graphs, the update sequence is part of the instance. However, the performance of algorithms may be affected greatly if an originally real-world update sequence is permuted randomly [13]. For this reason, we also consider five "shuffled" versions per real-world network, where the edge

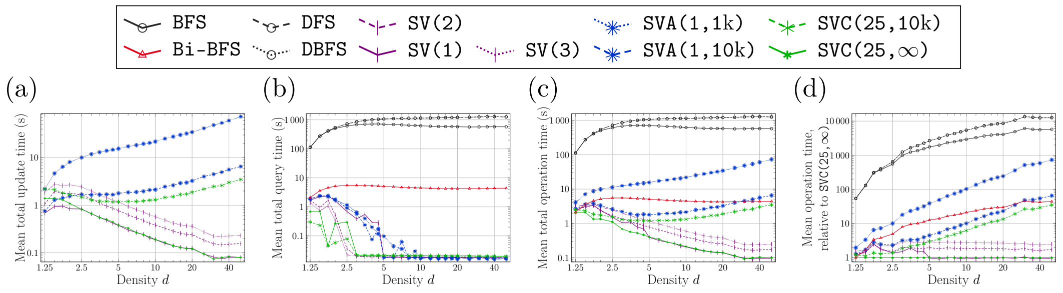

Figure 1 Random instances: $n = \sigma = 100\text{k}$ and equally many insertions, deletions, and queries.

insertions and deletions have been permuted randomly. We refer to the set of original instances as `konect` and to the modified ones as `konect-shuffled`. Removals of non-existing edges have been ignored in all instances. In each case, the updates are dominated by insertions.

Experimental Results

We ran the algorithms `SV`, `SVA`, and `SVC` with different parameters: For `SV`(k), we looked at $k = 1$, $k = 2$, and $k = 3$, which pick one, two, and three supportive vertices during initialization, respectively, and never reconsider this choice. We evaluate the variant that periodically picks new supportive vertices, `SVA`(k, c), with $k = 1$ and $c = 1\text{k}$, $c = 10\text{k}$, and $c = 100\text{k}$. Preliminary tests for `SVC` revealed $z = 25$ as a good threshold for the minimum SCC size on smaller instances and $z = 50$ on larger. The values considered for c were again 10k and 100k. `BiBFS` served as fallback for all *Supportive Vertices* algorithms. Except for random ER instances with $n = 10\text{m}$, all experiments also included `BFS`, `DFS`, and `DBFS`; to save space, the bar plots only show the result for the best of these three. We used `SES` as subalgorithm on random instances, and `SI` on real-world instances, in accordance with the experimental study for single-source reachability [13]. More plots are available in the full version of this paper [12].

ER Instances. We start by assessing the average performance of all algorithms by looking at their running times on random ER instances. For $n = 100\text{k}$ and equally many insertions, deletions, and queries, Figure 1 shows the mean running time needed to process all updates, all queries, and their sum, all operations, absolutely as well as the relative performances for all operations, where the mean is taken over 20 instances per density. As there are equally many insertions and deletions, the density remains constant. Note that all plots for random ER instances use logarithmic axes in both dimensions.

It comes as no surprise that `SV(2)` and `SV(3)` are two and three times slower on *updates*, respectively, than `SV(1)` (cf. **Figure 1a**). As their update time consists solely of the update times of their SSR data structures, they inherit their behavior and become faster, the denser the instance [13]. The additional work performed by `SVA`$(1, 1\text{k})$ and `SVA`$(1, 10\text{k})$, which re-initialized their SSR data structures 66 and six times, respectively, is plainly visible and increases also relatively with growing number of edges, which fits the theoretical (re-)initialization time of $\mathcal{O}(n + m)$. Computing the SCCs only initially, as `SVC`$(25, \infty)$ does, led to higher update times on very sparse instances due to an increased number of supportive vertices, but matched the performance of `SV(1)` for $d \geq 2.5$. As expected, re-running the SCC computation negatively affects the update time. In contrast to `SVA`$(1, 10\text{k})$, however, `SVC`$(25, 10\text{k})$ keeps a supportive vertex as long as it still represents an SCC, and thereby saves

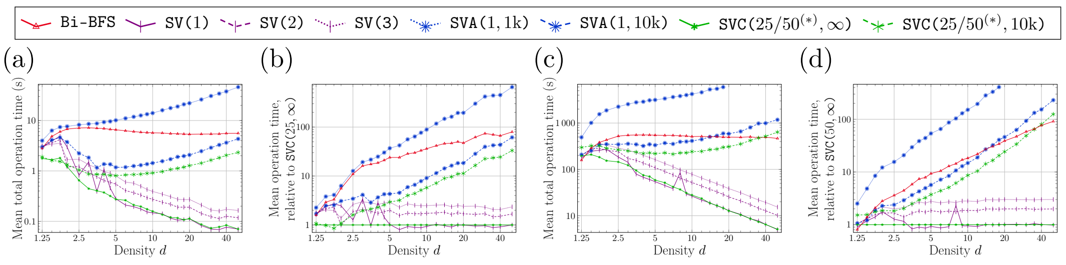

Figure 2 Random instances: $n = \sigma = 100$k and 50 % queries (a, b); $n = 10$m, $\sigma = 100$k, and equally many insertions, deletions, and queries (c, d).
$^{(*)}$ `SVC` parameter: 25 for $n = 100$k, 50 for $n = 10$m.

the time to destroy the old SSR data structures and re-initialize the new ones. Evidently, both `SVA` algorithms used a single supportive vertex for $d \geq 2.5$.

Looking at *queries* (cf. **Figure 1b**), it becomes apparent that `SVC(25, 10k)` can make use of its well-updated SCC representatives as supportive vertices and speed up queries up to a factor of 54 in comparison to `SVA` and `SV`. Up to $d = 3$, it also outperforms `SVC(25, ∞)`. For larger densities, the query times among all dynamic algorithms level up progressively and reach at least equality already at $d = 2.5$ in case of `SV(2)` and `SV(3)`, at $d = 5$ in case of `SV(1)`, and at $d = 10$ at the latest for all others. This matches a well-known result from random graph theory that simple ER graphs with $m > n \ln n$ are strongly connected with high probability [2]. The running times also fit our investigations into the mean percentage of queries answered during the different stages in query(·) by the algorithms: For $d = 2$, `SV(1)` could answer 80 % of all queries without falling back to `BiBFS`, which grew to almost 100 % for $d = 5$ and above. `SV(2)` answered even more than 95 % queries without fallback for $d = 2$, and close to 100 % already for $d = 3$. The same applied to `SVC(25, ∞)`, which could use SCC representatives in the majority of these fast queries. `SV(1)` and `SV(2)` instead used mainly observation **(O1)**, but also **(O2)** and **(O3)** in up to 10 % of all queries. As all vertices are somewhat alike in ER graphs, periodically picking new supportive vertices does not affect the mean query performance. In fact, `SV` and `SVA` are up to 20 % faster than `SVC` on the medium and denser instances, which can be explained by the missing overhead for maintaining the map of representatives. All *Supportive Vertices* algorithms process queries considerably faster than `BiBFS`. The average speedup ranges between almost 7 on the sparsest graphs in relation to `SVC(25, 10k)` and more than 240 in relation to `SV(1)` on the densest ones. The traditional static algorithms `BFS`, `DFS`, as well as the hybrid `DBFS` were *distinctly slower* by a factor of up to 31k (`BFS`) and almost 70k (`DFS`, `DBFS`) in comparison to `SV(1)`, and even 53 to 130 and 290 times slower than `BiBFS`.

In sum over *all operations*, if there are equally many insertions, deletions, and queries (cf. **Figures 1c, d**), `SVC(25, ∞)` and `SV(1)` were the *fastest algorithms* on all instances, where `SVC(25, ∞)` won on the sparser and `SV(1)` won on the denser instances. For $d = 1.25$, `BiBFS` was almost as fast, but up to 45 times slower on all denser instances. `SV(2)` and `SV(3)` could not compensate their doubled and tripled update costs, respectively, by their speedup in query time, which also holds for `SVC(25, 10k)`. `BFS`, `DFS`, and `DBFS` were between 54 and 13k times slower than `SVC(25, ∞)` and `SV(1)`, despite the high proportion of updates, and are therefore *far from competitive*.

We repeated our experiments with $n = 100$k and 50 % *queries* among the operations and equally many insertions and deletions, as well as with $n = 10$m and equal ratios of

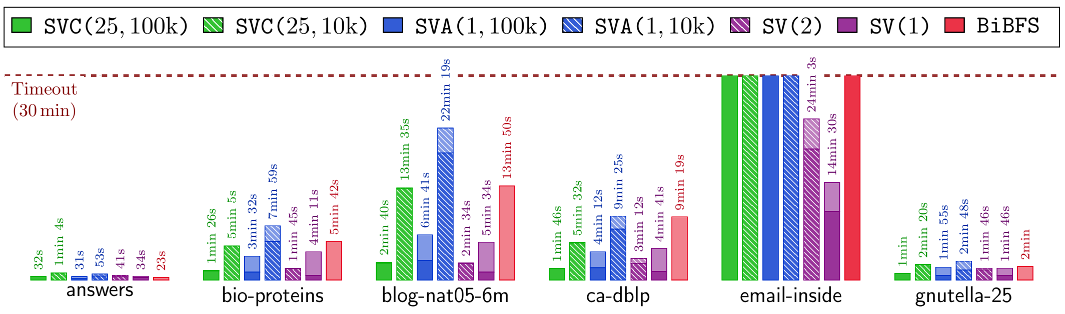

Figure 3 Total update (dark color) and query (light color) times on selected `kronecker-csize` instances.

insertions, deletions, and queries. The results, shown in **Figure 2**, *confirm* our findings above. In case of 50 % *queries*, a second supportive vertex as in `SV(2)` additionally stabilized the mean running time in comparison to `SV(1)`, up to $d = 5$ (cf. **Figures 2a, b**), but none of them could beat `SVC(25,`∞`)` on sparse instances. On denser graphs, `SV(1)` was again equally fast or even up to 20 % faster. As expected due to the higher ratio of queries, `BiBFS` lost in competitiveness in comparison to the above results and is between 1.6 and almost 80 times slower than `SVC(25,`∞`)` on dense instances. On the set of larger instances with $n = 10\text{m}$ (cf. **Figures 2c, d**), `SVA(1,1k)` reached the timeout set at 2 h on instances with $d \geq 20$. The *fastest* algorithms on average across all densities were again `SV(1)` and `SVC(50,`∞`)`. `BiBFS` won for $d = 1.25$, where it was about 20 % faster than `SVC(50,`∞`)` and 10 % faster than `SV(1)`. Its relative performance then deteriorated with increasing density up to a slowdown factor of 91. Except for $d = 1.25$, `SVC(50,`∞`)` outperformed `SV(1)` on very sparse instances and was in general also more stable in performance, as can be observed for $d = 8$: Here, `SV(1)` picked a bad supportive vertex on one of the instances, which resulted in distinctly increased mean, median, and maximum query times. On instances with density around $\ln n$ and above, `SV(1)` was slightly faster due to its simpler procedure to answer queries and also more stable than on sparser graphs.

In *summary*, `SVC` with $c = \infty$ clearly showed the *best and most reliable performance* on average, closely followed by `SV(1)`, which was *slightly faster* if the graphs were *dense*.

Kronecker Instances. In contrast to ER instances, stochastic Kronecker graphs were designed to model real-world networks, where vertex degrees typically follow a power-law distribution and the neighborhoods are more diverse. For this reason, the choice of supportive vertices might have more effect on the algorithms' performances than on ER instances. **Figure 3** shows six selected results for all dynamic algorithms as well as `BiBFS` on `kronecker-csize` instances with a query ratio of 33 %: Each bar consists of two parts, the lower, darker one depicts the total update time, the lighter one on top the total query time, which is comparatively small for most dynamic algorithms and therefore sometimes hardly discernible. In case that an algorithm reached the timeout, we only use the darker color for the bar. The description next to each bar states the total operation time. By and large, the picture is very similar to that for ER instances. On 13 and 14 out of the 20 instances, `BFS`/`DBFS` and `DFS`, respectively, did not finish within six hours. As on the previous sets of instances, these algorithms are *far from competitive*. The performance of `BiBFS` was ambivalent: it was the fastest on two instances, but lagged far behind on others. `SV(1)` and `SV(2)` showed the *best performance* on the *majority* of instances and were the *only ones* to finish within six hours on the largest instance of the set, email-inside. On some graphs, the total operation time of `SV(1)` was dominated by the query time,

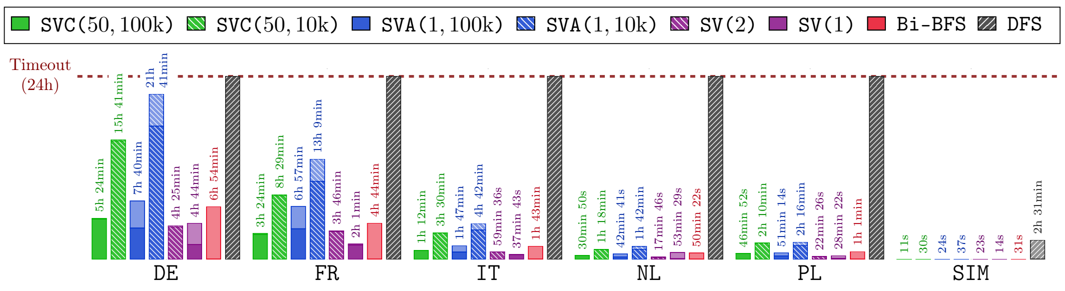

Figure 4 Total update (dark color) and query (light color) times on `konect` instances.

e.g., on bio-proteins, whereas `SV(2)` was able to reduce the total operation time by more than half by picking a second supportive vertex. However, `SVC(25, 100k)` was even able to outperform this slightly and was the *fastest* algorithm on half of the instances. As above, recomputing the SCCs more often (`SVC(25, 10k)`) or periodically picking new support vertices (`SVA(1, 1k)`, `SVA(1, 10k)`) led to a slowdown in general.

On `kronecker-growing`, `SV(1)` was the *fastest* algorithm on all but one instance. The overall picture is very similar.

Real-World Instances. In the same style as above, **Figure 4** shows the results on the six real-world instances with real-world update sequences, `konect`, again with 33 % queries among the operations. We set the timeout at 24 h, which was reached by `BFS`, `DFS`, and `DBFS` on all but the smallest instance. On the largest instance, `DE`, they were able to process only around 6 % of all updates and queries within this time. The *fastest algorithms* were `SV(1)` and `SV(2)`. If `SV(1)` chose the single support vertex well, as in case of `FR`, `IT`, and `SIM`, the query costs and the total operation times were low; on the other instances, the second support vertex, as chosen by `SV(2)`, could speed up the queries further and even compensate the cost for maintaining a second pair of SSR data structures. Even though the instances are growing and most vertices were isolated and therefore not eligible as supportive vertex during initialization, periodically picking new supportive vertices, as `SVA` does, did not improve the running time. `SVC(50, ∞)` performed well, but the extra effort to *compute the SCCs* and use their representatives as supportive vertices *did not pay off*; only on `SIM`, `SVC(50, ∞)` was able to outperform both `SV(1)` and `SV(2)` marginally.

Randomly permuting the sequence of update operations, as for the instance set `konect-shuffled`, did not change the overall picture.

5 Conclusion

Our extensive experiments on a diverse set of instances draw a somewhat surprisingly consistent picture: The most simple algorithm from our family, `SV(1)`, which picks a single supportive vertex, performed extremely well and was the fastest on a large portion of the instances. On those graphs where it was not the best, `SV(2)` could speed up the total running time by picking a second supportive vertex, i.e., the faster query time could compensate for the doubled update time. Additional statistical evaluations showed that already for sparse graphs, `SV(1)` and `SV(2)` answered a great majority of all queries in constant time using only its supportive vertices. Recomputing the strongly connected components of the graph in very large intervals and using them for the choice of supportive vertices yielded a comparatively good or marginally better algorithm on random instances, but not on real-world graphs.

The classic static algorithms BFS and DFS, which were competitive or even superior to the dynamic algorithms evaluated experimentally in previous studies, lagged far behind the new algorithms and were outperformed by several orders of magnitude.

References

1 Algora – a modular algorithms library. `https://libalgora.gitlab.io`.

2 B. Bollobás. *Random graphs*. Number 73 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2001.

3 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter Elementary Data Structures. MIT Press, 3rd edition, 2009.

4 C. Demetrescu and G. F. Italiano. Trade-offs for fully dynamic transitive closure on dags: Breaking through the $\mathcal{O}(n^2)$ barrier. *J. ACM*, 52(2):147–156, March 2005. `doi:10.1145/1059513.1059514`.

5 C. Demetrescu and G. F. Italiano. Mantaining dynamic matrices for fully dynamic transitive closure. *Algorithmica*, 51(4):387–427, 2008.

6 J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, April 1972. `doi:10.1145/321694.321699`.

7 R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, June 1962. `doi:10.1145/367766.368168`.

8 L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. `doi:10.4153/CJM-1956-045-5`.

9 D. Frigioni, T. Miller, U. Nanni, and C. Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *Journal of Experimental Algorithmics (JEA)*, 6:9, 2001.

10 A. Gitter, A. Gupta, J. Klein-Seetharaman, and Z. Bar-Joseph. Discovering pathways by orienting edges in protein interaction networks. *Nucleic Acids Research*, 39(4):e22–e22, November 2010.

11 A. V. Goldberg, S. Hed, H. Kaplan, R. E. Tarjan, and R. F. Werneck. Maximum flows by incremental breadth-first search. In *European Symposium on Algorithms*, pages 457–468. Springer, 2011.

12 K. Hanauer, M. Henzinger, and C. Schulz. Faster fully dynamic transitive closure in practice, 2020. `arXiv:2002.00813`.

13 K. Hanauer, M. Henzinger, and C. Schulz. Fully dynamic single-source reachability in practice: An experimental study. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 106–119, 2020. `doi:10.1137/1.9781611976007.9`.

14 M. Henzinger and V King. Fully dynamic biconnectivity and transitive closure. In *36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 664–672. IEEE, 1995.

15 M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *47th ACM Symposium on Theory of Computing*, STOC'15, pages 21–30. ACM, 2015.

16 T. Ibaraki and N. Katoh. On-line computation of transitive closures of graphs. *Information Processing Letters*, 16(2):95–97, 1983. `doi:10.1016/0020-0190(83)90033-9`.

17 G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986. `doi:10.1016/0304-3975(86)90098-8`.

18 G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28(1):5–11, 1988.

19 V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, page 81, USA, 1999. IEEE Computer Society.

20 V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *Journal of Computer and System Sciences*, 65(1):150–167, 2002. `doi:10.1006/jcss.2002.1883`.

21 V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proceedings of the 7th Annual International Conference on Computing and Combinatorics*, COCOON '01, page 268–277, Berlin, Heidelberg, 2001. Springer-Verlag.

22 I. Krommidas and C. D. Zaroliagis. An experimental study of algorithms for fully dynamic transitive closure. *ACM Journal of Experimental Algorithmics*, 12:1.6:1–1.6:22, 2008. `doi:10.1145/1227161.1370597`.

23 J. Kunegis. Konect: the Koblenz network collection. In *22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

24 F. Le Gall. Powers of tensors and fast matrix multiplication. In K. Nabeshima, K. Nagasaka, F. Winkler, and Á. Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303. ACM, 2014. `doi:10.1145/2608628.2608664`.

25 J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11:985–1042, March 2010. URL: `http://dl.acm.org/citation.cfm?id=1756006.1756039`.

26 J. Leskovec and R. Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

27 J. Łącki. Improved deterministic algorithms for decremental transitive closure and strongly connected components. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, page 1438–1445, USA, 2011. Society for Industrial and Applied Mathematics.

28 T. Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.

29 L. Roditty. A faster and simpler fully dynamic transitive closure. *ACM Trans. Algorithms*, 4(1), March 2008. `doi:10.1145/1328911.1328917`.

30 L. Roditty. Decremental maintenance of strongly connected components. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 1143–1150, USA, 2013. Society for Industrial and Applied Mathematics.

31 L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM Journal on Computing*, 37(5):1455–1471, 2008. `doi:10.1137/060650271`.

32 L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM J. Comput.*, 41(3):670–683, 2012. `doi:10.1137/090776573`.

33 L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM Journal on Computing*, 45(3):712–733, 2016.

34 P. Sankowski. Dynamic transitive closure via dynamic matrix inverse. In *45th Symposium on Foundations of Computer Science (FOCS)*, pages 509–517. IEEE, 2004.

35 Y. Shiloach and S. Even. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, 1981.

36 R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. `doi:10.1137/0201010`.

37 J. van den Brand, D. Nanongkai, and T. Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 456–480, 2019. `doi:10.1109/FOCS.2019.00036`.

38 S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, January 1962. `doi:10.1145/321105.321107`.

39 D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30(4):369–384, 1993.