Fully Dynamic Single-Source Reachability in Practice: An Experimental Study

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

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

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

Abstract

Given a directed graph and a source vertex, the *fully dynamic single-source reachability* problem is to maintain the set of vertices that are reachable from the given vertex, subject to edge deletions and insertions. It is one of the most fundamental problems on graphs and appears directly or indirectly in many and varied applications. While there has been theoretical work on this problem, showing both linear conditional lower bounds for the fully dynamic problem and insertions-only and deletions-only upper bounds beating these conditional lower bounds, there has been no experimental study that compares the performance of fully dynamic reachability algorithms in practice. Previous experimental studies in this area concentrated only on the more general all-pairs reachability or transitive closure problem and did not use real-world dynamic graphs.

In this paper, we bridge this gap by empirically studying an extensive set of algorithms for the single-source reachability problem in the fully dynamic setting. In particular, we design several fully dynamic variants of well-known approaches to obtain and maintain reachability information with respect to a distinguished source. Moreover, we extend the existing insertions-only or deletions-only upper bounds into fully dynamic algorithms. Even though the worst-case time per operation of all the fully dynamic algorithms we evaluate is at least linear in the number of edges in the graph (as is to be expected given the conditional lower bounds) we show in our extensive experimental evaluation that their performance differs greatly, both on generated as well as on real-world instances.

Keywords

Dynamic Graph Algorithms, Reachability, Algorithm Engineering

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

Many real-world problems can be expressed using graphs and in turn be solved using graph algorithms. 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. Another typical example is the OpenStreetMap road network, which is permanently subject to change as roads are built or (temporarily) closed, or simply because new information is added to the system by users. Given a concrete graph problem, computing a new solution for every change that occurs in the graph can be an expensive task on huge networks or where hardware resources are scarce, and ignores the previously gathered information on the instance under consideration. Hence, a whole body of algorithms and data structures for dynamic graphs has been discovered in the last decades. It is not surprising that dynamic algorithms and data structures are in most cases more difficult to design and analyze than their static counterparts.

Typically, dynamic graph problems are classified by the types of updates allowed. A problem is said to be fully dynamic if the update operations include insertions *and* deletions of edges. If only insertions are allowed, the problem is called incremental; if only deletions are allowed, it is called decremental.

One of the most basic questions that one can pose is that of reachability in graphs, i.e., answering the question whether there is a directed path between two distinct vertices. Already this simple problem has many applications such as in program analysis [22], spanning from compiler optimization to software security, or in the analysis of social or hyperlink networks—eg, whether somebody is a friend of a friend, relationship detection, or centrality measures. It also appears in computational biology, when analyzing metabolic or protein-protein interaction networks [8]. Additionally, it is a very important subproblem in a wide range of more complex (dynamic) algorithms such as in the computation of (dynamic) maximum flows [6, 5, 9], which in turn have manifold applications. However, state-of-the-art implementations typically run (slow) static breadth-first searches repeatedly to accomplish this task since there is no knowledge about the performance of more sophisticated algorithms in practice.

The single-source reachability problem has been extensively analyzed theoretically. The *fully dynamic single-source reachability* (*SSR*) problem is to maintain the set of vertices that are reachable from a given *source vertex*, subject to edge deletions and insertions. For the static version of the problem, i.e., when the graph does not change over time, reachability queries can be answered in constant time after linear preprocessing time by running, e.g., breadth-first search from the source vertex and marking each reachable vertex. This approach can be extended in the insertions-only case by using incremental breadth-first search so that each insertion takes *amortized* constant time and each query takes constant time. In the fully dynamic case, however, conditional lower bounds [13, 1] give a strong indication that no faster solution than the naive recomputation from scratch is possible after each change in the graph. There has been a large body of research on the deletions-only case [24, 11, 3], leading to a $\mathcal{O}(\log^4 n)$ [2] amortized expected time per deletion. However, to the best of our knowledge, there has been no prior experimental evaluation of fully dynamic *single-source* reachability algorithms.

In this paper, we attempt to start bridging this gap by empirically studying an extensive set of algorithms for the single-source reachability problem in the fully dynamic setting. In particular, we design several fully dynamic variants of well-known static approaches to obtain and maintain reachability information with respect to a distinguished source. Moreover, we modify existing algorithms that provide theoretical guarantees under the insertions-only or deletions-only setting to be fully dynamic. We then perform an extensive experimental evaluation on random as well as real-world instances in order to compare the performance of these algorithms. In addition, we introduce and assess different thresholds that trigger a recomputation from scratch to miti-

gate extreme update costs, which turned out to be very effective. Our results further show that making the insertions-only or deletions-only algorithms fully dynamic leads to faster algorithms than "dynamizing" static breadth-first or depth-first search.

2 Preliminaries

2.1 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 (*out-*/*in-*) *degree* of a vertex is its number of (out-/in-) edges. 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 *dynamic graph* is a directed graph G along with an ordered sequence of updates, which consist of edge insertions and deletions.

The paper deals with the *fully dynamic single-source reachability problem* (*SSR*): Given a directed graph and a source vertex s, answer reachability queries starting at s, subject to edge insertions and deletions.

2.2 Related Work

A whole body of algorithms [24, 10, 15, 11, 12, 3, 2, 14, 23] for *SSR* has been discovered in the last decades and has been complemented by several results on lower bounds [13, 1, 25]. In the incremental setting, an incremental breadth-first or depth-first search yields a total update time of $\mathcal{O}(m)$. The same update time can be achieved also in the decremental setting if the graph is acyclic [14]. For general graphs, the currently best decremental algorithm maintains reachability information in $\mathcal{O}(m \log^4 n)$ time [2]. In the fully dynamic setting, the fastest algorithm is randomized with one-sided error and uses dynamic matrix inverse, with a worst-case time of $\mathcal{O}(n^{1.575})$ per update and $\mathcal{O}(n^{0.575})$ per query [23]. Assuming the OMV conjecture, no algorithm for *SSR* exists with a worst-case update time of $\mathcal{O}(n^{1-\delta})$ and a worst-case query time of $\mathcal{O}(n^{2-\delta})$, for any $\delta > 0$ [13]. Moreover, a combinatorial *SSR* algorithm with a worst-case update or query time of $\mathcal{O}(n^{2-\delta})$ would also imply faster combinatorial algorithms for Boolean matrix multiplication and other problems [1, 25]. See Section A.1 for more details.

In extensive studies, Frigioni et al. [7] as well as Krommidas and Zaroliagis [16] have evaluated a huge set of algorithms for the more general fully dynamic all-pairs reachability problem experimentally on random dynamic graphs of size up to 700 vertices as well as two static real-world graphs with randomly generated update operations. They concluded that, despite their simple-mindedness, static breadth-first or depth-first search outperform their dynamic competitors on a large number of instances. There has also been recent development in designing algorithms that maintain a reachability index in the static setting [21, 26, 4, 27], which were evaluated experimentally [21] on acyclic random and real-world graphs of similar sizes as in this paper.

3 Algorithms

We implemented and tested a variety of deterministic, combinatorial algorithms. An overview is given in Table 1. Additionally, Table 2 subsumes the corresponding theoretical worst-case running times and space requirements. Not all of them are fully dynamic or even dynamic in their original form and have therefore been "dynamized" by us in a more or less straightforward manner. In this section, we provide a short description of these algorithms, their implementation, and the variants we considered. Each algorithm consists of up to four subroutines: initialize(), edgeInserted((u, v)), edgeDeleted((u, v)), and query(t), which define the algorithm's behavior during its initialization

Table 1: Algorithms and abbreviations overview.

Algorithm	Long name	Algorithm	Long name
`SDFS` / `CDFS` / `LDFS`	Static/Caching/Lazy DFS	`ES(`β`/`ρ`)`	Even-Shiloach
`SBFS` / `CBFS` / `LBFS`	Static/Caching/Lazy BFS	`MES(`β`/`ρ`)`	Multi-Level Even-Shiloach
`SI(R?/SF?/`ρ`)`	Simple Incremental	`SES(`β`/`ρ`)`	Simplified Even-Shiloach

phase, in case that an edge (u, v) is added or removed, and if it is queried whether a vertex t is reachable from the source, respectively. We distinguish three groups: The first group comprises algorithms that are based on static breadth-first and depth-first search with some improvements. Algorithms in the second group are based on a simple incremental algorithm that maintains an arbitrary, not necessarily height-minimal, reachability tree, and algorithms in the third group use Even-Shiloach trees and thus maintain a (height-minimal) breadth-first search tree. We did not implement (and extend to being fully dynamic) the more sophisticated deletions-only single-source reachability algorithms [11, 12, 3, 2] as they are very involved and maintain, e.g., a multi-level hierarchy of graphs and node separators, where Even-Shiloach trees appear only as sub-datastructures. Due to the resulting huge constants in worst-case time and space complexities, we expect them to perform much slower in practice. In the following, we assume an incidence list representation of the graph, i.e., each vertex has a list of incoming and outgoing edges.

3.1 Dynamized Static Algorithms

Depth-first search (*DFS*) and breadth-first search (*BFS*) are the two classic approaches to obtain reachability information in a static setting. Despite their simplicity, studies for all-pairs reachability [7, 16] report even their pure versions to be at least competitive with genuine dynamic algorithms and even superior on various instances. We consider three variants each: For our variants `SDFS` and `SBFS` (*Static DFS/BFS*), we do not maintain any information and start the pure, static algorithm for each query anew from the source. Thus, all work is done in query(·).

Second, we introduce a cache as a simple means to speedup queries for our variants `CDFS` and `CBFS` (*Caching DFS/BFS*). The cache contains reachability information for *all* vertices and is recomputed entirely in query(·) if it has been invalidated by an update. The rules for cache invalidation are as follows: An edge insertion is considered *critical* if it connects a reachable vertex to a previously unreachable vertex. Similarly, an edge deletion is *critical* if its head is `reachable`. The algorithms keep track of whether a critical insertion or deletion has occurred since the last recomputation. The cache is invalidated if either a critical insertion has occurred and the cached reachability state of a queried vertex t is `unreachable`, or if a critical deletion has occurred and the cached reachability state of t is `reachable`. Both algorithms may use initialize() to build their cache.

Finally, we also implemented lazy, caching variants `LDFS` and `LBFS` (*Lazy DFS/BFS*). In contrast to the former two, these algorithms only keep reachability information of vertices they have encountered while answering a query. As a vertex can only be assumed to be `unreachable` if the graph traversal has been exhaustive, the algorithms additionally maintain a flag `exhausted`. For query(t), the cached state of t is hence returned if t's cached state is `reachable` and no critical edge deletion has occurred. Otherwise, in case that there was no critical edge insertion and v's cached state is `unreachable`, the algorithm has to check the flag `exhausted`. If it is not set, the graph traversal that has been started at a previous query is resumed, thereby updating the cache, until either t is encountered or all reachable vertices have been visited. Then, the algorithm returns t's (cached) state. In all other cases, the cache is invalidated and the traversal must be started anew.

Table 2: Worst-case running times and space requirements.

Algorithm	Time			Space	
	Insertion	Deletion	Query	Permanent	Update
SBFS, SDFS	0	0	$\mathcal{O}(n+m)$	0	$\mathcal{O}(m)$
CBFS, CDFS, LBFS, LDFS	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n+m)$	$\mathcal{O}(n)$	$\mathcal{O}(m)$
SI(R?,SF?,ρ) ⌞ $\rho = 0$	$\mathcal{O}(n+m)$	$\mathcal{O}(n \cdot m)$ $\mathcal{O}(n+m)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(m)$
ES(β,ρ), MES(β,ρ) ⌞ $\beta \in \mathcal{O}(1) \vee \rho = 0$	$\mathcal{O}(n+m)$	$\mathcal{O}(n \cdot m)$ $\mathcal{O}(n+m)$	$\mathcal{O}(1)$	$\mathcal{O}(n+m)$	$\mathcal{O}(m)$
SES(β,ρ) ⌞ $\beta \in \mathcal{O}(1) \vee \rho = 0$	$\mathcal{O}(n+m)$	$\mathcal{O}(n \cdot m)$ $\mathcal{O}(n+m)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(m)$

3.2 Reachability-Tree Algorithms

In a pure incremental setting, i.e., without edge deletions, an algorithm that behaves like LDFS or LBFS, but updates its cache on edge insertions rather than queries, can answer queries in $\mathcal{O}(1)$ time and spends only $\mathcal{O}(n+m)$ in total for all edge insertions, i.e., its amortized time for an edge insertion is $\mathcal{O}(1)$. We refer to this algorithm as SI (*Simple Incremental*) and describe various options to make it fully dynamic. For every vertex $v \in V$, SI maintains a flag reachable$[v]$, which is used to implement query(v) in constant time, as well as a pointer treeEdge$[v]$ to the edge in the reachability tree whose head is v. More specifically, the algorithm implements the different operations as follows:
initialize(): The algorithm traverses the graph using BFS starting from s and sets reachable$[v]$ and treeEdge$[v]$ for each vertex $v \in V$ accordingly.
edgeInserted((u,v)): If u, but not v was reachable before, update reachable and treeEdge of all vertices that can be reached from v and were unreachable before by performing a BFS starting at v.
edgeDeleted((u,v)): If treeEdge$[v] = (u,v)$, the deletion of (u,v) requires to check and update all vertices in the subtree rooted at v. We consider two basic options: Updating the stored reachability information or recomputing it entirely from scratch. For the former, we first identify a list $\mathcal{L}$ of vertices whose reachability is possibly affected by the edge deletion, which comprises all vertices in the subtree rooted at v and is obtained by a simple preorder traversal. Their state is temporarily set to unknown and their treeEdge pointers are reset. Then, the reachability of every vertex w in $\mathcal{L}$ is recomputed by traversing the graph by a backwards BFS starting from w until a reachable ancestor x is found or the graph is exhausted. If w is reachable, the vertices on the path from x to w are added to the reachability tree using the path's edges as tree edges. If w is unreachable, so must be all vertices encountered during the backwards traversal. In both cases, this may, thus, reduce the number of vertices with state unknown. Optionally, if w is reachable, the algorithm may additionally start a forward BFS traversal from w to update the reachability information of all vertices with status unknown in $\mathcal{L}$ that are reachable from w. Moreover, $\mathcal{L}$ can be processed in order either as constructed or reversed. Independently of this choice, the worst-case running time is in $\mathcal{O}(|\mathcal{L}| \cdot (|\mathcal{L}| + m))$, as vertices in $\mathcal{L}$ may be traversed $\mathcal{O}(|\mathcal{L}|)$ times by the backwards BFS. Recomputing from scratch, the second option, requires $\mathcal{O}(n+m)$ worst-case update time.

Thus, our implementation of SI takes three parameters: two boolean flags R (negated: $\overline{\text{R}}$) and SF (negated: $\overline{\text{SF}}$), specifying whether $\mathcal{L}$ should be processed in reverse order and whether a forward search should be started for each re-reachable vertex, respectively, as well as a ratio $\rho \in [0,1]$ indicating that if $\mathcal{L}$ contains more than $\rho \cdot n$ elements, the reachability information for *all* vertices is recomputed from scratch.

3.3 Shortest-Path-Tree Algorithms

In 1981, Even and Shiloach [24] described a simple decremental connectivity algorithm for undirected graphs that is based on the maintenance of a BFS tree and requires $\mathcal{O}(n)$ amortized update time. Such a tree is also called Even-Shiloach tree or ES tree for short. Henzinger and King [10] were the first to observe that ES trees immediately also yield a decremental algorithm for *SSR* on directed graphs with the same amortized update time if the source is used as the tree's root. We extend this data structure to make it fully dynamic and consider various variants.

For every vertex $v \in V$, an ES tree maintains its BFS level $\mathtt{l}[v]$, which corresponds to v's distance from s, as well as an ordered list of in-edges $\mathcal{E}^-[v]$. To efficiently manage this list in the fully dynamic setting, the algorithm additionally uses an index of size $\mathcal{O}(m)$ that maps each edge (u, v) to its position in $\mathcal{E}^-[v]$. If v is reachable, its *tree edge* in the BFS tree is the edge with tail at level $\mathtt{l}[v] - 1$ whose index i is the smallest in $\mathcal{E}^-[v]$ (invariant). The algorithm stores the tree edge's index in $\mathcal{E}^-[v]$ as $\mathtt{e}[v]$. If v is unreachable, $\mathtt{l}[v] = \infty$ (invariant). A reachability query query(t) can thus be answered in $\mathcal{O}(1)$ by testing whether $\mathtt{l}[t] \neq \infty$.

initialize(): The ES tree is built by a BFS traversal starting from the source. In doing so, $\mathcal{E}^-[v]$ is populated for each vertex v in the order in which the edges are encountered. Thus, after the initialization, $\mathtt{e}[v] = 0$. The update operations are implemented as follows.

edgeInserted((u, v)): Update the data structure in worst-case $\mathcal{O}(n + m)$ time by starting a BFS from v and checking for each vertex that is encountered whether either its level or, subordinately, its parent index can be decreased.

edgeDeleted((u, v)): If (u, v) is v's tree edge, the algorithm tries to find a substitute edge. To this end, v is added to an initially empty FIFO-queue Q containing vertices whose tree edge and, if necessary, whose level has to be newly determined. Vertices in Q are processed one-by-one as follows: For each vertex w, the index $\mathtt{e}[w]$ is increased until it either points to an edge with tail at level $\mathtt{l}[w]-1$ or $\mathcal{E}^-[w]$ is exhausted. In the latter case, if $\mathtt{l}[w]+1 < n$, w's level is increased by one, $\mathtt{e}[w]$ is reset to zero, and all children of w in the BFS tree as well as w itself are added to Q. Otherwise, w is unreachable and $\mathtt{l}[w] := \infty$. This operation has a worst-case running time of $\mathcal{O}(n \cdot m)$.

In view of this large update cost, we again introduce an option to alternatively recompute the BFS tree from scratch. We use two parameters to control the algorithm's behavior: a factor ρ that limits the number of vertices that may be processed in the queue to $\rho \cdot n$ as well as an upper bound β on how often a vertex may be (re-)inserted into the queue before the update operation is aborted and a recomputation is triggered. We refer to this algorithm as ES (*Even-Shiloach*). Observe that if the algorithm recomputes immediately, i.e., if $\rho = 0$, or each vertex may be processed in Q only a constant number of times i.e., if $\beta \in \mathcal{O}(1)$, the worst-case theoretical running time is only $\mathcal{O}(n+m)$.

We also implemented a variation of ES that sets the tree edge of a vertex w in the queue directly to the first edge in $\mathcal{E}^-[w]$ whose tail has the lowest level and updates $\mathtt{l}[w]$ accordingly, which avoids the immediate re-insertion of w into the queue. More precisely, while iterating through $\mathcal{E}^-[w]$, as realized by increasing $\mathtt{e}[w]$, this variation keeps track of the minimum level $l_{\min}$ and the corresponding index $e_{\min}$ of an edge's tail encountered thereby. If $\mathtt{e}[w]$ reaches $|\mathcal{E}^-[w]|$, i.e., no incoming edge with tail at level $\mathtt{l}[w]-1$ has been found, $\mathtt{e}[w]$ is set to 0 and the search continues until $\mathtt{e}[w]$ attains the value it had when removed from Q. Then, $\mathtt{l}[w]$ is set to $l_{\min}+1$, $\mathtt{e}[w] = e_{\min}$, and, if $\mathtt{l}[w]$ has increased, all children of w in the BFS tree, but not w itself, are added to Q. As vertices may skip several levels in one step, we refer to this version of ES as MES (*Multi-Level Even-Shiloach*).

We also consider an even further simplification of ES, SES (*Simplified Even-Shiloach*), which does no longer maintain an ordered list of in-edges for each vertex v and hence also no index $\mathtt{e}[v]$. Instead, it stores for each reachable vertex a direct pointer to its tree edge in the BFS tree. For each vertex w in Q, SES simply iterates over all in-edges in arbitrary order and sets w's tree edge to one whose tail has minimum level. If this increases w's level, all children of

w in the BFS tree are added to `Q`. Both `MES` and `SES` take the same two parameters as `ES` to control when to recompute the data structure from scratch.

4 Experiments

4.1 Environmental Conditions and Methodology

We evaluated the performance of all algorithms described in Section 3 with all available parameters on both generated and real-world instances. All algorithms were implemented[1] in C++17 and compiled with GCC 7 using full optimization (`-O3 -march=native -mtune=native`). Experiments were run on a machine with two Intel Xeon E5-2643 v4 processors clocked at 3.4 GHz and 1.5TB of RAM under Ubuntu Linux 18.04 LTS with kernel 4.15. Each experiment was assigned exclusively to one core.

For each algorithm and graph, we measured the time spent during initialization as well as for each insertion, deletion, and query. From these, we obtained the total insertion time, total deletion time, total update time, and total query time as the respective sums. For the smaller random instances, we ran each experiment three times and use the medians of the aggregations for the evaluation to counteract artifacts of measurement and accuracy.

In the following, we use k and m as abbreviations for $\times 10^3$ and $\times 10^6$, respectively.

4.2 Instances

Random Instances. To assess the average performance of our algorithms, we generated a set of smaller random directed graphs according to the Erdős-Renyí model $G(n, m)$ with $n = 100$k vertices and $m = d \cdot n$ edges, where $d \in [1.25 \dots 50]$, in each case along with a random sequence of operations σ consisting of edge insertions, edge deletions, as well as reachability queries. In the same fashion, we generated a set of larger instances with $n = 10$m vertices and $m = d \cdot n$ edges. For insertions, we drew pairs of vertices uniformly at random from V, allowing also for parallel edges. For deletions and reachability queries, each edge or vertex, respectively, was equally likely to be chosen. For a fixed source vertex, we tested sequences of $\sigma = 100$k operations, where insertions, deletions, and queries appear in batches of ten, but are processed individually by the algorithms. We evaluated different proportions of the three types of operations.

Kronecker Instances. Reachability plays an important role in the analysis of social networks, whose structures differ greatly from that of Erdős-Renyí graphs, e.g., in terms of degree distribution. Our test instances therefore additionally include stochastic Kronecker instances [18], which were shown to model the structure of such networks very well. We generated two sets containing 20 instances each, using the `krongen` tool that is part of the SNAP software library [20] and the estimated initiator matrices given in [18] that correspond to real-world networks. To obtain dynamic graphs, we generated a sequence of different *snapshot graphs* for each initiator matrix, computed the differences between two subsequent instances, and simulated an update sequence by applying them in random order. In the first set, we used sequences of ten graphs that were generated in 17 iterations with up to ≈130k vertices each, whereas in the second, the graphs in each sequence were generated with increasing number of iterations, starting from five up to 17, which resulted in instances having around 30 vertices initially and again up to ≈130k in the end. We refer to the first set as `kronecker-csize` and to the second as `kronecker-growing`. For each dynamic graph, we used the ten vertices with highest out-degree in their respective initial graph as sources. All instances in `kronecker-csize` have densities between 0.7 and 16.4. Their update sequences consist of equally many insertions and deletions, whose lengths range between 1.6m and 702m. In `kronecker-growing`, the densities vary between 0.9 and 16.4. There are 282k to 82m updates, 66 % to 75 % of which are insertions.

[1] We plan to release the code publicly.

Real-World Instances from KONECT. Our set of test instances is complemented by a collection of real-world dynamic networks, which also includes real-world update sequences. For algorithms that maintain a reachability tree, the latter is especially of interest, as the selection and order of edge insertions and deletions may affect the amount of work required immensely. We used all six directed, dynamic instances available from the Koblenz Network Collection KONECT [17], a collection of real-world graphs from various application scenarios. The graphs are given as a list of edge insertions and deletions, each of which is assigned a timestamp, and model the hyperlink structure between Wikipedia articles for six different languages. Hyperlink networks are a variant of social networks, where reachability information is used, e.g., to detect dependencies or topical clusters. For our evaluation, the edge insertions and deletions with the smallest timestamp form the initial graph, and all further updates are grouped by their timestamp. We set the source vertex to be the tail of the first edge with minimum timestamp. Our instances have between 100k (simple English) and 2.2m vertices (French) and from initially less than five up to 747k to 24.5m edges, which result from between 1.6m and 86m update operations, consisting of both edge insertions and deletions. We refer to these instances as `FR`, `DE`, `IT`, `NL`, `PL`, and `SIM`.

To see whether differences in the algorithms' performance are rather due to the structure of the graphs or the order of updates, we generated five new, "shuffled" instances per language by randomly assigning new timestamps to the update operations, which we refer to as *shuffled KONECT*. As for the original instances provided by KONECT, we ignored removals of non-existing edges.

Real-World Instances from SNAP. Additionally, we use a collection of 122 snapshots of the computer network describing relationships in the CAIDA Internet Autonomous System, which is made available via the Stanford Large Network Dataset Collection SNAP [19]. We built a dynamic, directed graph `AS-CAIDA` with $n = 31$k and $m = 73$k to 113k from this collection by using the differences between two subsequent snapshots as updates. Edges are directed from provider to customer and there is a pair of anti-parallel edges between peers and siblings. We obtained ten instances from this graph by choosing one of the ten vertices with highest out-degree, respectively, as source.

Table A.3 lists the detailed numbers for all real-world instances. In each case, the updates are dominated by insertions, which constitute 51 % for `AS-CAIDA` and 68 % to 76 % for KONECT. The average density varies between 3.2 (`AS-CAIDA`) and 7.8 (`IT`).

4.3 Experimental Results

4.3.1 Random Graphs

For $n = 100$k, we generated 20 graphs per density $d = \frac{m}{n}$ along with a sequence of 100k operations, where edge insertions, edge deletions, and queries were equally likely. In consequence, the density of each dynamic graph remains more or less constant during the update sequence. The timeout was set to one hour. Figure 1 depicts the results, which we will discuss in the following. Note that the plots use logarithmic axis in both dimensions.

Relative Performances within Groups (Figures 1a–d). For the discussion of the results, we group the algorithms as in Section 3. The **first group** consists of the six **dynamized static algorithms** `SBFS`, `SDFS`, `CBFS`, `CDFS`, `LBFS`, and `LDFS`. Recall that all work is done in query(·) here, which is why we evaluate them based on their mean total query time. Figure 1a shows the relative performance of this algorithm group compared to `LBFS`, which was the *best algorithm* on average over all densities and for each density always seven to 16 times faster on average than the "pure" static algorithms `SBFS` and `SDFS`. Up to a density of 4.5, `LBFS` is beaten by `LDFS`, however, the performance gap between `LBFS` and `LDFS` increases at least linearly as the graphs become denser. The eager caching versions `CBFS` and `CDFS` show similar performance to their lazy counterparts on sparse graphs, but then deteriorate exponentially compared to the latter and eventually even fall behind the pure

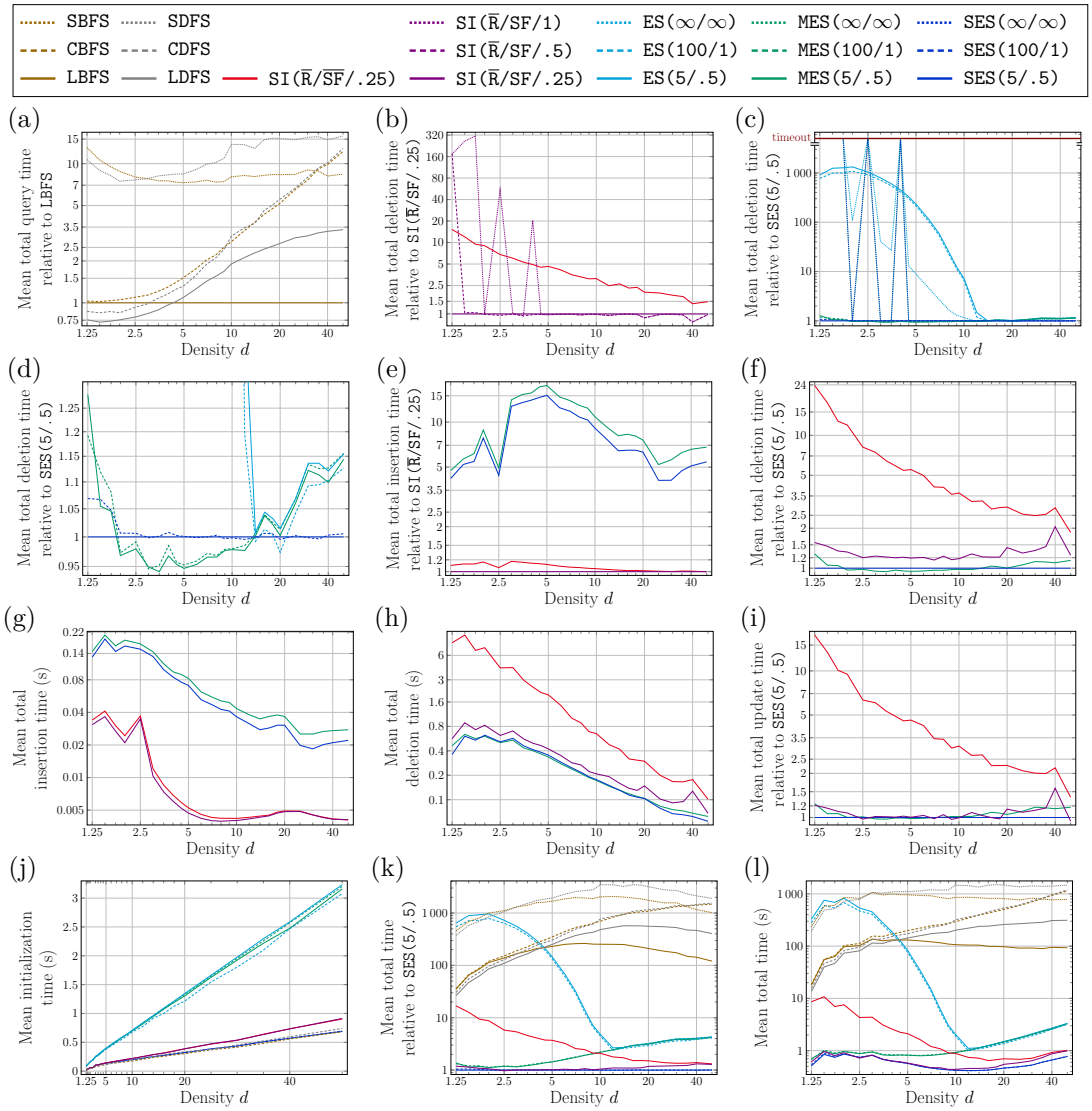

Figure 1: Results on random instances with $n = 100\,000$, $\sigma = 100\,000$, and $d \in [1.25, \ldots, 50]$.

static variants SBFS and SDFS, respectively. *To summarize, the algorithms based on DFS are only faster than their BFS-based counterparts on sparser instances and distinctly slower on denser ones.*

The **second group** of algorithms consists of the fully dynamic variants of the **simple incremental** algorithm SI. These algorithms only differ in their implementation of edgeDeleted(·) and, thus, we evaluate them on their mean deletion time. We tested different combinations of the boolean flags R and SF along with different values for the recomputation threshold ρ. *One main observation is that, regardless of ρ, the algorithms SI(R̄/SF/ρ) were faster than the algorithms using other combinations of the flags, but the same value ρ*, where the worst-performing was SI(R̄/SF̄/ρ). If the flags R? and SF? were fixed, smaller values for ρ showed better performance than larger, except for extremely small ones. Recall that if ρ is zero, the algorithm always discards its current reachability tree and recomputes it from scratch using BFS, whereas if ρ is one, it always reconstructs a reachability tree. Hence, ρ may be seen as a means to control outliers that necessitate the re-evaluation of the reachability of a large number of vertices. To keep the number of variants

manageable, Figure 1b only shows the relative mean total deletion time of SI with four different parameter sets: $\overline{\text{R}}$/SF with $\rho = 0.25$, $\rho = 0.5$, and $\rho = 1$, respectively, and $\overline{\text{R}}/\overline{\text{SF}}$ with $\rho = 0.25$. The *fastest algorithm* on average across all densities in this set was SI($\overline{\text{R}}$/SF/.25), which is therefore also used as reference. The same algorithm with disabled forward search, i.e., SI($\overline{\text{R}}/\overline{\text{SF}}$/.25), was up to a factor of around 16 slower on sparse graphs. As the graphs become denser, this factor decreases exponentially down to less than 1.5 for graphs having $d = 40.0$ and above. The reason for this will be discussed with Figure 1h. SI($\overline{\text{R}}$/SF/.5) and SI($\overline{\text{R}}$/SF/1) show similar performance as SI($\overline{\text{R}}$/SF/.25) for densities of at least 1.5 and 2.0, respectively, however with extreme spikes at $d = 2.5$ and $d = 4.0$ if $\rho = 1$, which are caused by few instances with enormous cost for re-establishing the reachability tree. For $d = 2.5$, e.g., SI($\overline{\text{R}}$/SF/1) needed around 10 min for one specific edge deletion operation on one graph, whereas the maximum deletion time on all other instances was less than 150 ms. The total deletion time hence was less than 1 s on 19 instances and around 10 min on the 20th, which resulted in a mean total deletion time of 33.7 s for SI($\overline{\text{R}}$/SF/1) on graphs with density 2.5. By contrast, the mean total deletion time of SI($\overline{\text{R}}$/SF/.25) on these 20 instances was 619 ms. The other spikes can be explained similarly. *In conclusion, low values for ρ can effectively control outliers and speed up the average deletion time by factors of up to* 307.

The **third group** of algorithms comprises those based on **ES trees**: ES, MES, and SES. We tested each of them with different values for the parameters β and ρ. Here, both parameters serve to limit excessive update costs that occur when either the levels of a smaller set of vertices in the ES tree increase multiple times (β) or a large set of vertices is affected (ρ). They turned out to be very useful. We tested three parameter sets: An early abortion of the update process and recomputation with $\beta = 5$ and $\rho = 0.5$, a late variant with $\beta = 100$ and $\rho = 1$, and finally $\beta = \infty$ and $\rho = \infty$, which does not impose any limits. Similar as in case of SI, the algorithms only differ in their implementation of edgeDeleted($\cdot$). Figure 1c reports the mean total deletion time relative to the (on average) *best algorithm* in this set, SES(5/.5). For sparse graphs, the ES algorithms were up to approximately 1400 times slower than SES(5/.5). This factor drops super-exponentially as the graphs become denser and reaches a value of around 1 near $d = 12$. The unlimited variants showed an even worse performance on graphs up to a density of 4.0 with several timeouts, but a performance similar to, or, in case of ES, even better one than their limited versions for denser graphs. In all cases, the timeouts occurred on six instances with $d = 1.5$, four with $d = 1.75$, and one with $d = 2.5$ and $d = 4.0$, respectively. For $d = 1.25$, ES(∞/∞) timeouted four times, whereas the MES and SES variants only once. Apart from one exception, all timeouts were caused by a single deletion operation that took more than one hour, in some cases even more than five.

Differences between the limited versions of MES and SES are barely observable on this scale. Figure 1d zooms in on the values of interest for these algorithms. Evidently, SES(5/.5) outperformed MES(5/.5) both on very sparse instances up to $d = 1.75$ as well as on denser ones from $d = 14$ and onward. In the middle range, it was less than 6 % slower than MES(5/.5). Recall that in contrast to SES, MES stores information about the incoming edges of a vertex. However, for very sparse as well as denser instances, the additional knowledge available to MES seemingly cannot outweigh the increased workload that comes with the maintenance of this information: In the former case, the list of in-edges is short and therefore scanned very quickly in SES, whereas in the latter, a replacement tree edge with tail on the same level can be expected to be found very early in SES's scanning process. *To summarize, for both SES and MES, the variants that are more reluctant to recompute from scratch performed slightly worse than their respective counterparts. The ES algorithms were almost always outperformed by MES and SES.*

Update Performances (Figures 1e–i). Next, we compare the relative performances of the SI and the ES/MES/SES algorithm classes using SI($\overline{\text{R}}$/SF/.25), SI($\overline{\text{R}}/\overline{\text{SF}}$/.25), MES(5/.5), and SES(5/.5) as representatives. Figure 1e depicts the mean average total insertion times. Despite identical implementation, SI($\overline{\text{R}}$/SF/.25) was slightly faster than SI($\overline{\text{R}}/\overline{\text{SF}}$/.25) on sparser instances, which

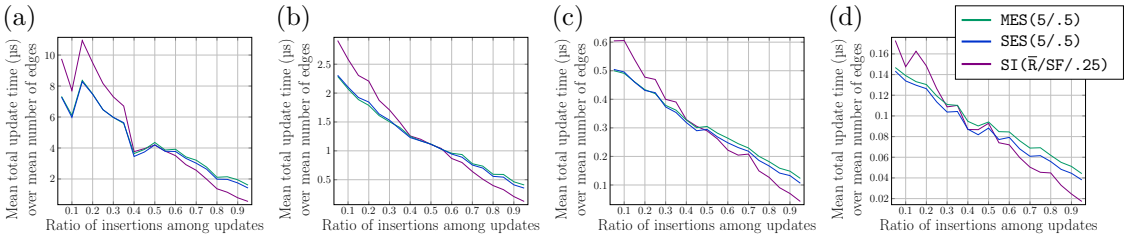

Figure 2: Mean total update times in μs relative to the mean average number of edges for varying ratios of insertions on random instances with $n = 100\,000$, $\sigma = 100\,000$, and initial density $d = 2.5$ (a), $d = 5$ (b), $d = 10$ (c), and $d = 20$ (d).

may be due to structural differences in their reachability trees. `MES(5/.5)` and `SES(5/.5)` were four to approximately 16 times slower than `SI(R̄/SF/.25)`, where the maximum was reached at a density of 5.0. These experimental results conform with the theoretical performance analysis of `SI`, which yields a "perfect" amortized update time of $\mathcal{O}(1)$ in the incremental setting. `MES(5/.5)` is slightly slower than `SES(5/.5)` due to the additional information it maintains. The overall situation is inverted in case of deletions, as Figure 1f shows. Here, `MES(5/.5)` and `SES(5/.5)` outperformed both `SI(R̄/SF/.25)` and `SI(R̄/S̄F̄/.25)`, the latter even by a factor of almost 24 on very sparse instances. `SI(R̄/SF/.25)` was 15 % to 100 % slower on average than `SES(5/.5)`.

These findings suggest that `SI(R̄/SF/.25)` would be the best choice among these algorithms unless the proportion of edge deletions is markedly high. However, insertions and deletions are not equally costly, as Figures 1g and 1h demonstrate. The best and worst mean total running times for insertions were roughly by a factor of 50 faster than for deletions. Moreover, they show that the effort to process an update decreased at least exponentially as the density increases. The reason for this observation is twofold: First, the probability that a newly inserted edge will be part of the reachability tree or that an edge of the current reachability tree is deleted diminishes as the density grows. This holds especially for the algorithms of the second group, which do not care about the distance from the source vertex, and where the reduction in the total insertion time is pronounced. Second, if a deletion of a tree edge really occurs, a replacement edge is usually not "too far". The latter also speeds up `SI(R̄/S̄F̄/.25)`'s process of handling edge deletions, as the relatively costly (and numerous) backwards breadth-first searches terminate quickly. Figure 1i depicts the relative mean total update times, where insertions and deletions occur with equal probability. As deletions are distinctly more time-consuming than insertions—`SES(5/.5)` and `MES(5/.5)` spent $\approx$ 70–85 %, `SI(R̄/SF/.25)` even $\approx$ 94–99 % of the update time on deletions—*`SES(5/.5)` showed the best performance on average over all densities.* Again, `MES(5/.5)` was slower on very sparse and slightly denser instances by up to about 20 %. `SI(R̄/SF/.25)`'s performance was roughly similar to `MES(5/.5)`'s, however with a largest deviation of 58 % from `SES(5/.5)`'s at $d = 40.0$.

The **initialization time**, as shown in Figure 1j, was as expected and is discussed in more detail in Section A.2.

Overall Performances. **Figures 1l and 1k** depict the mean total running time if insertions, deletions, and queries occur with equal probability. The fastest dynamized static algorithm, `LBFS`, was clearly outperformed by `SI(R̄/SF/.25)`, `MES(5/.5)`, and `SES(5/.5)` on all densities. For sparser graphs up to $d = 4.0$, however, the lazy and caching variants were faster than `ES`. On dense instances, where the update costs decrease rapidly, the initialization time begins to show through for `SI` and the `ES` family. *The `SES` algorithms performed best in these experiments, with `SES(5/.5)` being the overall fastest on average.*

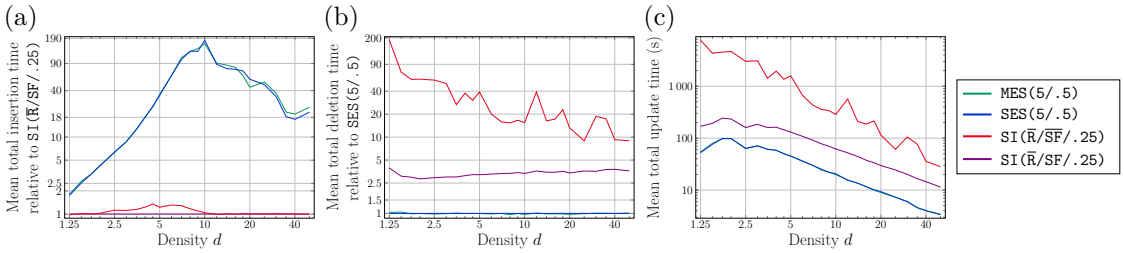

Figure 3: Results on random instances with $n = 10\text{m}$, $\sigma = 100\,000$, and $d \in [1.25, \ldots, 50]$.

Ratios of Insertions, Deletions, and Queries (Figure 2). We next investigate whether and how the picture changes if the proportion of insertions and deletions varies. Taking up on the observation that the SI algorithms were considerably faster on insertions than MES and SES, but slower on deletions, we compare the performance of the fastest of each of them, i.e., SI(R̄/SF/.25), MES(5/.5), and SES(5/.5) on random instances with $n = 100\text{k}$ vertices, different initial densities $d \in \{2.5, 5, 10, 20\}$, and $\sigma = 100\text{k}$. We sampled ten graphs per density. As unequal ratios of insertions and deletions change the density of the graphs over time, Figure 2 shows the mean total update time divided by the average number of edges. As expected, MES(5/.5), and SES(5/.5) outperformed SI(R̄/SF/.25) for low ratios of insertions, whereas the opposite holds if there are many insertions among the updates. *The threshold is around* 50 % *for all densities.* MES(5/.5) was similarly fast as SES(5/.5) if the proportion of deletions was high (and d is small), and became relatively slower as the ratio of insertions grew.

In our setting, all dynamized static algorithms were clearly inferior. We expected a performance increase if queries occur either very rarely or, if a cache is used, very frequently. We reviewed this assumption experimentally and found it confirmed. *However, none of the dynamized static algorithms could compete with the dynamic ones.* See Section A.3 for details.

Large Graphs (Figure 3). We repeated our experiments on larger graphs with $n = 10\text{m}$ vertices for the algorithms MES, SES, and SI. Figure 3 shows the mean total insertion time relative to SI(R̄/SF/.25), the total deletion time relative to SES(5/.5), as well as the absolute mean total update time. As for the instances with $n = 100\text{k}$, the update time was dominated heavily by the deletion time and decreased with growing density. The mean total update time relative to SES(5/.5) here almost equals the deletion time, which is shown together with further plots in Figure A.7. SES and MES with parameters 100/1 were almost identical to their more restricted counterparts and are therefore not shown. As before, SI outperformed MES and SES for insertions, whereas the latter two were faster than SI for deletions. *In both cases, the picture is similar to that for the smaller instances, however, the speedup factor has increased markedly.* In total, SI(R̄/SF/.25) was 2.5 to 10 times slower than the *best algorithms* MES and SES, which in turn performed almost identically.

4.3.2 Kronecker Instances

So far, we only assessed the algorithms' performance on random graphs generated according to the Erdős-Renyí model. Kronecker instances mimic real-world networks and hence exhibit a different structure. The results for the kronecker-csize graphs are shown in Figure 4. *On all instances, SES(5/.5) outperformed the other algorithms*, but was closely followed by MES(5/.5), whereas SI(R̄/SF/.25) was two to 15.2 times slower and SI(R̄/SF̄/.25) with slowdown factors of 6.9 to 57.5 was far from being competitive. Similar to the random instances, at least 71 % of the update time was spent on deletions, with one exception (email-inside, 51 %). Despite their higher insertion rate, the results for kronecker-growing are similar (cf. Figure A.8). *All in all, the picture is consistent with that on random instances.*

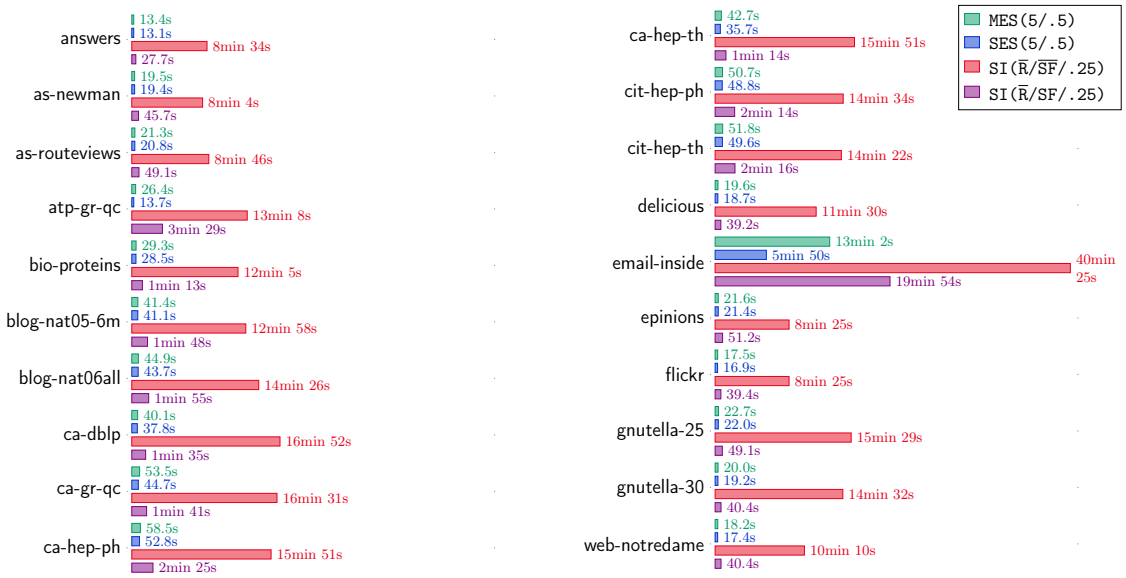

Figure 4: Mean update times on `kronecker-csize` instances, taken over ten different sources per instance.

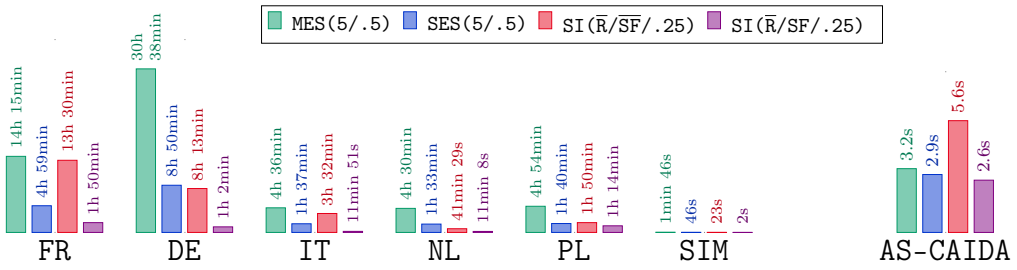

Figure 5: Update times on real-world instances from KONECT and SNAP.

4.3.3 Real-World Graphs

We evaluated the algorithms `MES`, `SES`, and `SI` also on real-world graphs that come with real-world update sequences. Figure 5 shows the update times for the KONECT instances as well as the mean update time of the SNAP instances. On all instances, the algorithms spent more than 89 % of the update time on deletions. On all instances, `SI(R̄/SF/.25)` here distinctly *outperforms* all competitors, followed with considerable distance by `SES(5/.5)`, which is in turn faster than `MES(5/.5)` by several factors. On `AS-CAIDA`, which has a mostly random update sequence, the lead of `SI(R̄/SF/.25)` is clearly less, but still visible.

The overall picture did not change for the shuffled KONECT instances with updates in random order, as depicted in Figure A.9. However, the speedup of `SI(R̄/SF/.25)` in comparison to `SES(5/.5)` decreased visibly in general, from up to 23 to a maximum of less than seven. The performance ratio of `MES(5/.5)` and `SES(5/.5)` remained constant. As the graphs at each point in time can be assumed to have similar characteristics as the Kronecker instances, these results demonstrate that the order of the updates (random or not) influences the performance of `SI(R̄/SF/.25)` and `SES(5/.5)`, but it can only partially explain that the former performs better on real-world graphs than on Kronecker and random graphs. Since deletions are significantly slower than insertions, we investigated the percentage of "expensive" deletions, i.e., deletions that change the reachability tree. For `SI(R̄/SF/.25)`, this number is at most 13 % on the KONECT graphs, but up to 33 % on Kro-

necker graphs with comparable density. For `SES(5/.5)`, however, this number is 19 % and 29 %, respectively. As the deletion time is much higher for `SI` than for `SES`, this can explain the difference in performance on real-world vs. Kronecker graphs. A reason for the relatively small change percentage in real-world graphs for `SI(R̄/SF/.25)` might be that the distribution of lifetimes of the edges is different in real-world and Kronecker graphs. In the latter, the probability that an edge exists in two subsequent snapshot graphs is very low, implying that the lifetime of every edge is relatively small, in contrast to edges representing hyperlinks between articles, as in the KONECT graphs.

5 Conclusion

The simplified Even-Shiloach algorithm, `SES`, with parameters `5/.5` showed the best performance on all instances except for the real-world dynamic graphs, where it was outperformed by the fully dynamic version of the simple incremental algorithm, `SI`, with parameters `R̄/SF/.25`. However, `SES` was in particular superior in handling edge deletions, which heavily dominated the update costs across all tested sets of instances. All algorithms benefitted considerably from introducing recomputation thresholds. Breadth-first search and depth-first search, even with enhancements, were unable to compete with the dynamic algorithms, irrespective of the proportion of queries. The impact of degree distribution on the algorithms' performance remains unclear.

In a nutshell: We recommend to use `SI(R̄/SF/.25)` on real-world networks with long-living edges or if the ratio of insertions is distinctly above 50 %, and otherwise `SES(5/.5)`.

References

[1] A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, FOCS '14, pages 434–443. IEEE, 2014.

[2] A. Bernstein, M. Probst, and C. Wulff-Nilsen. Decremental strongly-connected components and single-source reachability in near-linear time. In *Proceedings of the 51st Annual ACM Symposium on Theory of Computing*, STOC '19, 2019.

[3] S. Chechik, T. D. Hansen, G. F. Italiano, J. Łącki, and N. Parotsidis. Decremental single-source reachability and strongly connected components in $\tilde{O}(m\sqrt{n})$ total update time. In *57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 315–324. IEEE, 2016.

[4] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 193–204. ACM, 2013.

[5] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, Apr. 1972.

[6] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[7] 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.

[8] 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, 11 2010.

[9] 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.

[10] 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.

[11] M. Henzinger, S. Krinninger, and D. Nanongkai. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *46th ACM Symposium on Theory of Computing*, pages 674–683. ACM, 2014.

[12] M. Henzinger, S. Krinninger, and D. Nanongkai. Improved algorithms for decremental single-source reachability on directed graphs. In *Automata, Languages, and Programming*. Springer, 2015.

[13] 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.

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

[15] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Symposium on Foundations of Computer Science (FOCS)*, pages 81–89. IEEE, 1999.

[16] 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.

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

[18] 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, Mar. 2010.

[19] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[20] 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.

[21] F. Merz and P. Sanders. Preach: A fast lightweight reachability index using pruning and contraction hierarchies. In A. S. Schulz and D. Wagner, editors, *European Symposium on Algorithms*, pages 701–712, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

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

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

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

[25] V. V. Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In *51st Symposium on Foundations of Computer Science (FOCS)*, pages 645–654, 2010.

[26] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1601–1606. ACM, 2013.

[27] H. Yıldırım, V. Chaoji, and M. J. Zaki. Grail: a scalable index for reachability queries in very large graphs. *The VLDB Journal—The International Journal on Very Large Data Bases*, 21(4):509–534, 2012.

A Appendix

A.1 Related Work

In an incremental setting, where edges may only be inserted, but never are deleted, a total update time of $\mathcal{O}(m)$ for m insertions can be achieved by an incremental breadth-first or depth-first search starting from the source vertex. For a long time, the best algorithm to handle a series of m edge deletions and no insertions required a total update time of $\mathcal{O}(mn)$ and actually solved the more general all-pairs shortest path problem. The algorithm is due to Even and Shiloach [24, 10, 15] and maintains a breadth-first tree under edge deletions. It is widely known as *ES tree*. Recently, Henzinger et al. [11, 12] broke the $\mathcal{O}(mn)$ time barrier by giving a probabilistic algorithm with an expected total update time of $\mathcal{O}(mn^{0.9+o(1)})$. Shortly thereafter, Chechik et al. [3] improved this result further by presenting a randomized algorithm with $\tilde{\mathcal{O}}(m\sqrt{n})$ total update time. Only lately, Bernstein et al. [2] showed that reachability information in the decremental setting can be maintained in $\mathcal{O}(m \log^4 n)$ total expected update time. Whereas these algorithms all operate on general graphs, Italiano [14] observed that a running time of $\mathcal{O}(m)$ may indeed be achieved also in the decremental setting if the input graph is acyclic. Finally, if both edge insertions and deletions may occur, Sankowski's algorithms [23] for transitive closure imply a worst-case per-update running time of $\mathcal{O}(n^{1.575})$ for the fully dynamic single-source reachability problem.

On the negative side, Henzinger et al. [13] showed that unless the Online Matrix-Vector Multiplication problem can be solved in time $\mathcal{O}(n^{3-\varepsilon})$, $\varepsilon > 0$, no algorithm for the fully dynamic single-source reachability problem exists with a worst-case update time of $\mathcal{O}(n^{1-\delta})$ and a worst-case query time of $\mathcal{O}(n^{2-\delta})$, $\delta > 0$. Furthermore, if there is a combinatorial, fully dynamic s-t reachability algorithm with a worst-case running time of $\mathcal{O}(n^{2-\delta})$ per update or query, then there are also faster combinatorial algorithms for Boolean matrix multiplication and other problems, as shown by Abboud and Vassilevska Williams [1] and Williams and Vassilevska Williams [25], respectively.

In extensive studies, Frigioni et al. [7] as well as Krommidas and Zaroliagis [16] have evaluated a huge set of algorithms for the more general fully dynamic all-pairs reachability problem experimentally on random dynamic graphs of size up to 700 vertices as well as two static real-world graphs with randomly generated update operations. They concluded that, despite their simple-mindedness, static breadth-first or depth-first search outperform their dynamic competitors on a large number of instances. There has also been recent development in designing algorithms that maintain a reachability index in the static setting [21, 26, 4, 27], which were evaluated experimentally [21] on acyclic random and real-world graphs of similar sizes as in this paper.

A.2 Initialization time on Random Instances

Even though it is of less importance if the operation sequences are long, we take a brief look at the initialization time. The algorithms are split into three groups here: Whereas `SBFS`, `SDFS`, `LBFS`, and `LDFS` do not use this phase, all other algorithms traverse the graph once and build up their data structures. `CBFS`, `CDFS`, `SI`, and `SES` reserve and access $\mathcal{O}(n)$ space, but `ES` and `MES` need to setup $\mathcal{O}(n + m)$ space, which is clearly reflected in the running time, as Figure 1j shows. Note that Figure 1j does *not* use logarithmic scales.

A.3 Updates vs. Queries

All dynamized static algorithms were clearly inferior to their competitors on random instances with $n = 100$k if all types of operations occurred with equal probability, which corresponds to a proportion of queries of $\frac{1}{3}$. However, we expect a relative performance increase if either queries occur either very rarely or very frequently, where the latter naturally only applies to those algorithms that use a cache. We review this assumption experimentally by examining the performance of `CBFS`, `CDFS`, `LBFS`, and `LDFS` in comparison to `SI(R̄/SF/.25)`, `MES(5/.5)`, and `SES(5/.5)` for

varying ratios of queries among the operations. We did not include SBFS and SDFS, as LBFS and LDFS are always at least as fast. We again sampled ten instances with $n = 100$k vertices for each density $d \in \{2.5, 5, 10, 20\}$, in each case along with $\sigma = 100$k operations. To keep the density of the graphs constant, insertions and deletions occur with equal probabilities. Figure A.6 depicts the mean total operation times. Although the results confirm our assumption, none of the dynamized static algorithms can compete with the dynamic ones, neither for sparse nor for denser graphs.

A.4 Additional Tables and Plots

Table A.3: Number of vertices n, initial, average, and final number of edges m, $\overline{m}$, and M, average density $\frac{\overline{m}}{n}$, total number of updates δ with percentage of additions δ_+, and query success rate of real-world instances.

Instance	n	m	$\overline{m}$	M	$\frac{\overline{m}}{n}$	δ	δ_+	success
FR	2.2m	3	13.0m	24.5m	5.9	59.0m	71 %	39.7 %
DE	2.2m	4	16.7m	31.3m	7.6	86.2m	68 %	43.4 %
IT	1.2m	1	9.3m	17.1m	7.8	34.8m	75 %	52.0 %
NL	1.0m	1	5.7m	10.6m	5.7	20.1m	76 %	42.3 %
PL	1.0m	1	6.6m	12.6m	6.6	25.0m	75 %	42.4 %
SIM	100k	2	401k	747k	4.0	1.6m	73 %	39.7 %
AS-CAIDA	31k	73k	99.9k	113k	3.2	1.4m	51 %	69 %
FR_SHUF	2.2m	4.0	16.4m	30.4m	7.5	53.1m	79 %	51.6 %
DE_SHUF	2.2m	3.8	22.6m	41.1m	10.3	76.4m	77 %	61.7 %
IT_SHUF	1.2m	3.8	10.9m	20.5m	9.1	31.4m	83 %	62.8 %
NL_SHUF	1.0m	3.8	6.7m	12.6m	6.7	18.1m	85 %	57.5 %
PL_SHUF	1.0m	3.6	7.9m	14.9m	7.9	22.7m	83 %	55.8 %
SIM_SHUF	100k	5.6	476k	892k	4.8	1.6m	80 %	42.5 %

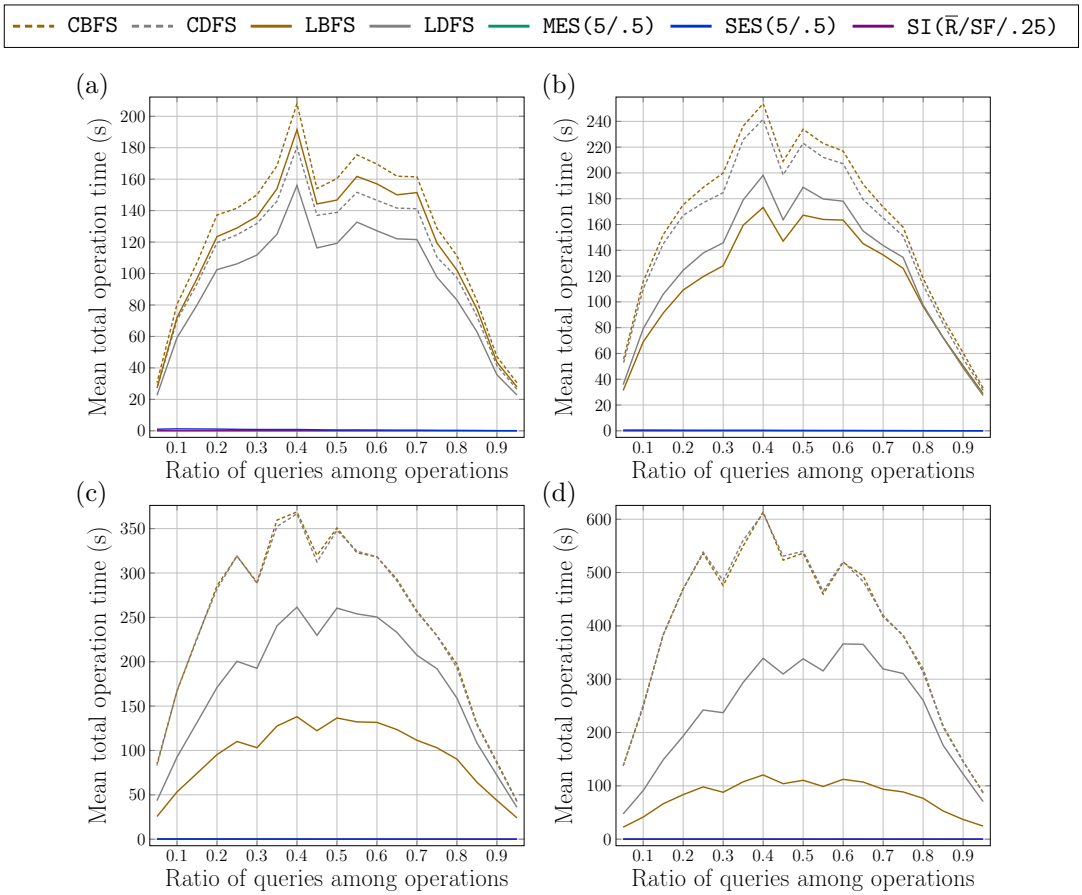

Figure A.6: Mean total operation times in seconds for varying ratios of queries and equal ratio of additions and deletions on random instances with $n = 100\,000$, $\sigma = 100\,000$, and initial density $d = 2.5$ (a), $d = 5$ (b), $d = 10$ (c), and $d = 20$ (d).

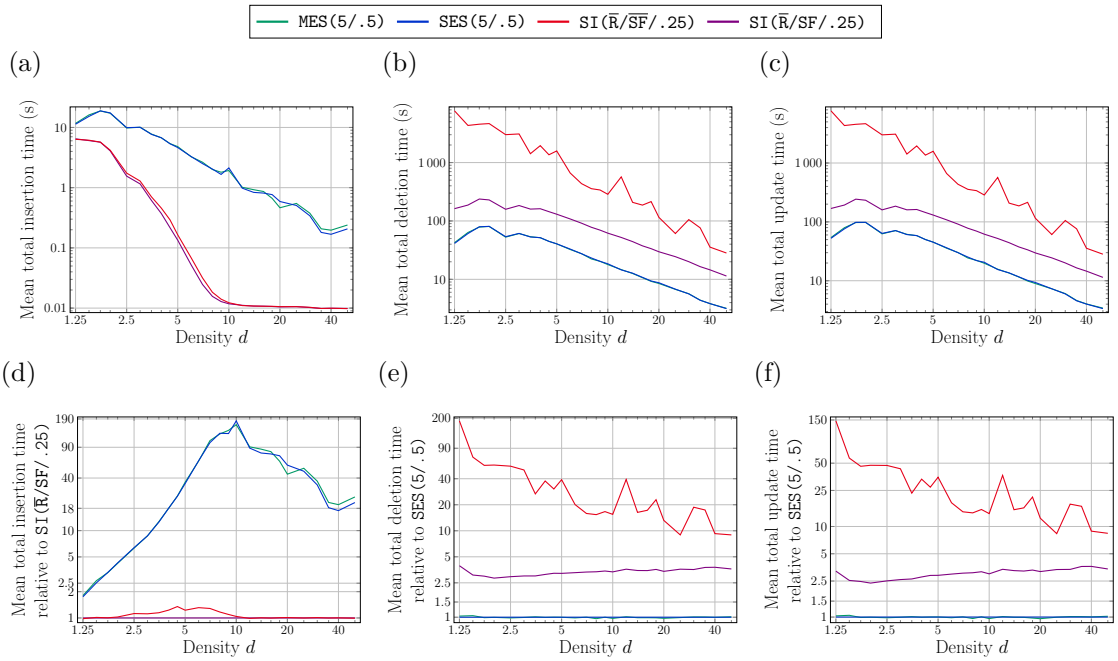

Figure A.7: Results on random instances with $n = 10\text{m}$, $\sigma = 100\,000$, and $d \in [1.25, \ldots, 50]$.

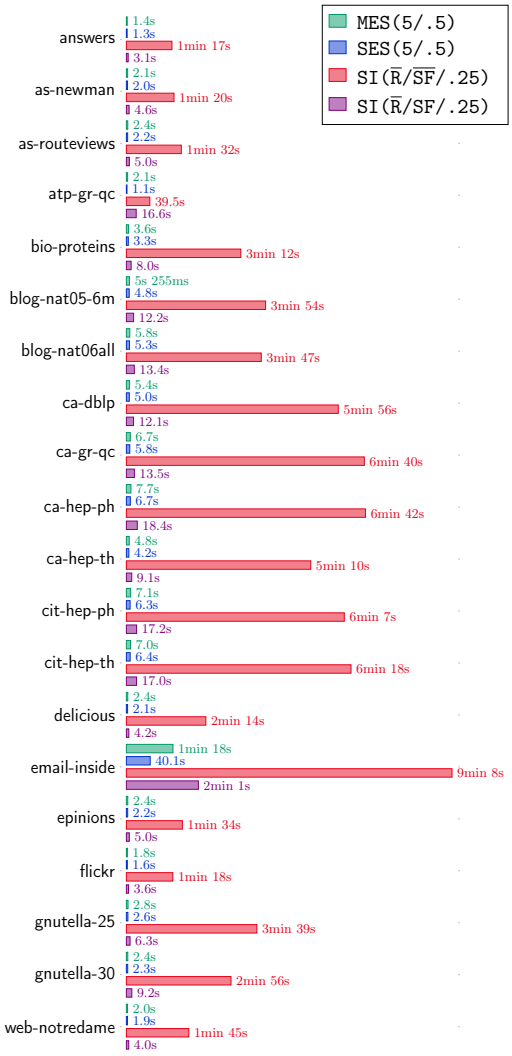

Figure A.8: Mean update times on `kronecker-growing` instances, taken over ten different sources per instance.

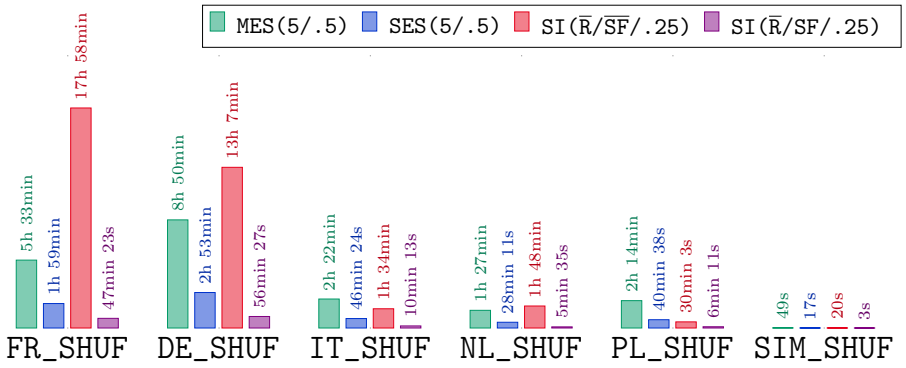

Figure A.9: Update times on shuffled real-world instances from KONECT.